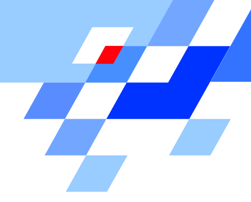
Universität Dortmund
Fachbereich Informatik
Lehrstuhl für Software-Technologie



MEMO Nr. 150

Abschlussbericht der PG 444
Eclipse Framework for Editing Complex Three-Dimensional Software Visualizations

Ernst-Erich Doberkat Alexander Fronk Jens Schröder (Hrsg.)

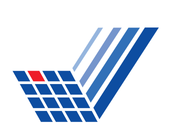
Armin Bruckhoff
Stephan Eisermann
Kai Gutberlet
Michél Kersjes
André Kupetz
Christian Mocek
Michael Nöthe
Michael Pflug
René Schönlein
Semih Sevinç
Daniel Unger
Sven Wenzel
Jan Wessling

August 2004

Internes Memorandum des
Lehrstuhls für Software-Technologie
Prof. Dr. Ernst-Erich Doberkat
Fachbereich Informatik
Universität Dortmund
Baroper Straße 301

D-44227 Dortmund

ISSN 0933-7725



Abschlussbericht der PG444

Eclipse Framework for Editing Complex Three-Dimensional Software Visualizations

Lehrstuhl für Software-Technologie
Universität Dortmund

Dieses Dokument wurde verfasst von
Armin Bruckhoff, Stephan Eisermann, Kai Gutberlet,
Michél Kersjes, André Kupetz, Christian Mocek, Michael Nöthe,
Michael Pflug, René Schönlein, Semih Sevinç, Daniel Unger,
Sven Wenzel, Jan Wessling

Projektleitung: Alexander Fronk, Jens Schröder

Inhaltsverzeichnis

4 Beschreibung des Frameworks

5 Fazit

6 Anhang

Abbildungsverzeichnis

TEIL 1

Einführung in die Projektgruppe *EFFECTS*

KAPITEL 1

Einleitung

Alexander Fronk, Jens Schröder

Visualisierung und grafische Methoden sind in der Softwaretechnik weit verbreitet. In den meisten Fällen werden dabei zweidimensionale Grafiken eingesetzt, wie es im Ingenieurswesen seit langer Zeit üblich ist. Der große Vorteil von zweidimensionalen Zeichnungen ist, dass sie alleine mit Papier und Bleistift erstellt werden können.

Die Dreidimensionalität bietet eine Reihe von Vorteilen, insbesondere wenn man die Reduktion auf Papier und Bleistift als alleinige Werkzeuge aufgibt:

- *Transparenz von Objekten*, die das Innere von Objekte visuelle zugreifbar macht
- *Tiefeneindruck*, der im Zweidimensionalen nicht existiert
- *Anordnung im Raum*
- *Bewegung von Objekten*, wie etwa die Rotation von Cone-Trees

Heutzutage werden in den Ingenieurswissenschaften oft dreidimensionale CAD-Systeme eingesetzt, wohingegen bei der Visualisierung großer Softwaresysteme die dritte Dimension nur selten genutzt wird. Im Gegensatz zu den Produkten, die im klassischen Ingenieursbereich konstruiert werden, ist Software abstrakt, da sie keine physikalische Manifestation hat. Es ist daher eine Herausforderung, geeignete dreidimensionale visuelle Konzepte zu finden.

In einer Reihe von Arbeiten wurde am Lehrstuhl für Software-Technologie ein System zur dreidimensionalen Visualisierung von statischen Relationen zwischen Java-Klassen entwickelt. In dem System wurden verschiedene Visualisierungstechniken wie Cone-Trees oder Information-Cubes in Verbindung mit Federmodelle für eine graphbasierte Visualisierungsmetapher eingesetzt (siehe Abb. 1.1).

Das bisher am Lehrstuhl für Software-Technologie entwickelte System bietet ausschließlich eine Visualisierung der Softwarestruktur an, eine Manipulation der visualisierten Struktur ist hingegen nicht möglich. Daher ist es wünschenswert, ein neues System zu konstruieren, das zusätzlich die graphische Manipulation der dreidimensionalen Visualisierung unterstützt, d.h. wir benötigen einen graphischen Editor für dreidimensionale Visualisierungen. Da bei dieser Manipulation immer auch die visualisierte Software (in unserem Fall Java-Programme) geändert wird, wird der Editor in die Java-Entwicklungsumgebung Eclipse eingebettet. Eclipse bietet sich durch seine moderne und elegante PlugIn-Struktur als erweiterbare Entwicklungsumgebung an. Als Schnittstelle für die dreidimensionale Darstellung soll die Java3D-API von Sun verwendet werden, um unabhängig von plattformspezifischen Bibliotheken zur dreidimensionalen Darstellung zu sein.

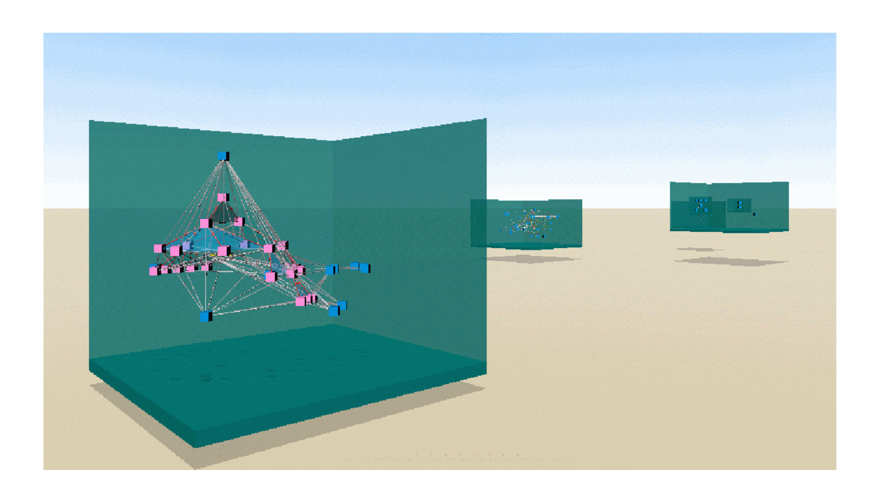


Abbildung 1.1.: dreidimensionale Darstellung einer Klassenstruktur

Die bisher am Lehrstuhl für Software-Technologie verwendete Graphmetapher zur Visualisierung bietet den Ausgangspunkt für die dreidimensionale Darstellung im Editor. Es sind aber natürlicherweise weitere Visualisierungstechniken und -metaphern denkbar, wie etwa hyperbolische Räume. Daher soll ein Framework konzipiert und realisiert werden, um unabhängig von einer bestimmten Metapher und einem festen Satz von Visualisierungstechniken Editoren zur dreidimensionalen Visualisierung von Softwarestrukturen einfach und elegant entwickeln zu können.

Es soll ein Editorframework für die dreidimensionale Darstellung und Manipulation von Softwarestrukturen konzipiert und realisiert werden. Informationen müssen im dreidimensionalen Raum sinnvoll und geeignet angeordnet werden, um Softwarestrukturen angemessen wiederzugeben. Dazu müssen verschieden Techniken miteinander kombiniert werden: Zur dreidimensionalen Darstellung soll die Java3D-API Verwendung finden; das Werkzeug soll als PlugIn der Entwicklungsumgebung Eclipse realisiert werden, da diese die zur Entwicklung des Frameworks benötigten Basisfunktionalitäten bereitstellt.

Das Framework soll es ermöglichen, komfortabel Werkzeuge, die jeweils alternative dreidimensionale Visualisierungentechniken und Metaphern zur Darstellung von Softwarestrukturen einsetzen, zu entwickeln.

Zusätzlich kann über das Ziel hinaus eine Validierung des generischen Ansatzes des Editorframeworks erfolgen, in dem weitere Notationen, Visualisierungstechniken oder -metaphern auf der Basis des Frameworks konzipiert und realisiert werden. Es können spezifische Aspekte der Projektgruppenarbeit als Diplomarbeiten vergeben werden.

KAPITEL 2

Geplantes Vorgehen

Alexander Fronk, Jens Schröder

Die Projektgruppenarbeit kann grob in folgende Phasen aufgeteilt werden: Einarbeitung, Anforderungsanalysephase und Konstruktionsphase mit eXtreme Programming, die durch eine angemessene Dokumentation und ein Fachgespräch ergänzt werden. Folgende Abschnitte erläutern die einzelnen Phasen im Detail.

2.1 Einarbeitung

In Form von Seminarvorträgen durch die Projektgruppenteilnehmer wird die Projektgruppe an die zu lösende Aufgabe herangeführt. Dies dient der Aneignung des nötigen Fachwissens. Die Einarbeitung erfolgt in folgende Themenbereiche und Problemfelder:

- Softwaretechnische Entwurfsnotationen
- dreidimensionale Visualisierungstechniken
- Design Patterns
- eXtreme Programming (XP)
- Graphische Editoren
- Visuelle Sprachen
- Eclipse
- Java3D-API

Neben einer inhaltlichen Einarbeitung hat die Projektgruppe die Gelegenheit, sich in die technische Arbeitsumgebung einzufinden und an ihre Bedürfnisse anzupassen.

2.2 Anforderungsanalyse

In dieser Phase wird die Projektgruppe die spezifischen Anforderungen an einen graphischen Editor für dreidimensionale Visualisierung herausarbeiten. Ein Ziel dabei ist es, eine geeignete Systemmetapher zu entwickeln, die als Voraussetzung für einen XP-basierten Konstruktionsprozess benötigt wird.

2.3 Konstruktion

Die Entwicklung des Editorframeworks in der Konstruktionsphase folgt dem XP-Ansatz. Auf Grund der ganzheitheitlichen Teamorientierung und der Eigenverantwortlichkeit der einzelnen Entwickler, die auch die selbstständige Planung der auszuführenden Tätigkeiten umfasst, bietet sich XP als Entwicklungsprozess für diese Projektgruppe an.

Dem XP-Ansatz folgend ergibt sich eine inkrementelle Entwicklung mit vielen kleineren Releases, bei der die frühen Releases einen eher prototypischen Charakter haben, die dann zu einen vollständigen System evolvieren.

2.4 Berichte

Die gesamten Arbeiten werden jeweils durch einen *Zwischen*- und einen *Abschlussbericht* dokumentiert.

Der Zwischenbericht dokumentiert die Ergebnisse des ersten Semesters, insbesondere die Anforderungsanalyse und die ersten Releases des zu erstellenden Systems.

Der Abschlussbericht wird den gesamten Projektverlauf festhalten. Die Ergebnisse der einzelnen Phasen werden vorgestellt und bewertet.

2.5 Exkursion und Fachgespräch

Den Abschluss der Projektgruppe bildet eine Exkursion zu einem dem Projektgruppenthema angemessenen Ziel.

Daneben wird ein Fachgespräch stattfinden, in dem die Projektgruppenteilnehmer den Fachbereich über den Ablauf und die Ergebnisse der Projektgruppe informieren. Dieses Fachgespräch wird im Rahmen des Diplomanden- und Doktorandenseminars des LS 10 stattfinden.

2.6 Zeitlicher Ablauf

Folgender zeitlicher Ablauf ist geplant:

1. Semester (16 Wochen) (Oktober bis Februar)

    - Einarbeitungsphase (3 Wochen)
    - Erste Releases erstellen mit dem Ziel, technologische Kenntnisse zu erwerben (11 Wochen):
        - Eclipse-PlugIn (Manipulation des Syntaxbaums)
        - Java3D (graphische Notation als Basisprimitive)

    Release 1 Technische Erfahrungen sammeln mit PlugIn-Schnittstelle von Eclipse und der Java3D-API:

- PlugIn, dass eine Menge von Klassen als Würfel darstellt. Zusätzlich müssen in dem Diagramm folgende Regeln gelten:
  - Die Vaterklasse muss oberhalb der Sohnklasse angeordnet werden
  - Klassen, die zu einem Paket gehören, müssen gruppiert angeordnet werden
- In dem Diagramm sind somit eine Reihe von impliziten Strukturen enthalten:
  - Domainentitäten, die betrachtet werden: Klassen, Pakete, Vererbungsrelation, Paketenthaltenseinsbeziehungen
  - Diagrammentiäten, die verwendet werden:
    -
  - Regeln, wie die Domainentitäten im Diagram dargestellt werden:
    - Klassen werden als Würfel visualisiert
    - die Vererbungsrelation wird auf eine entsprechende Anordnung entlang der z-Achse abgebildet
    - Paketenthaltenseinsbeziehung werden durch eine Clusterbildung in x-y-Ebene dargestellt

Release 2 Durch Anwenden von Refactoring das PlugIn aus Release 1 als Framework realisieren
- konfigurierbare Hotspots für die wesentlichen Aspekte eines Diagramms (xml-basiert, Eclipse-PlugIn-Schnittstelle als Vorbild):
  - Domainhotspots
  - Diagrammhotspots
  - Regeln zur Anordnung

Release 3 Interaktionsdiagramme (Kombination aus Sequenz- und Kollaborationsdiagramme) als neuen Diagrammtyp entwickeln, um:
- eine Reifeprüfung des Frameworks durchzuführen
- eine Verbesserung durch Anwenden von Reengineering und Refactoring zu erzielen

- Zwischenbericht (2 Wochen)

2. Semester (15 Wochen) (April bis Juli)

- Erstellung weiterer Releases zur funktionalen Vervollständigung des PlugIns (12 Wochen)

Release 4 Erstellen eines Editor-PlugIns für eine Kombination eines Klassen- und Paketdiagramms: Interaktion- und Manipulationsmöglichkeiten realisieren. Es wird folgende Notation umgesetzt:
- Klassen werden als Würfel dargestellt
- Schnittstellen werden als Kugeln dargestellt
- Pakete werden als Information Cubes dargestellt

  - Vererbungshierarchien werden als Cone Trees dargestellt
  - Assoziationen werden als Röhren dargestellt

Release 5 PlugIn aus Release 4 um Funktionalität anreichern:
  - aus dem Diagramm Code erzeugen
  - Bedienelemenete in die Eclipse-Oberfläche integrieren
  - Darstellungen überarbeiten

- Abschlussbericht (3 Wochen)

2.7 Vorgehensmodell

Als Vorgehensmodell wird der Ansatz des eXtreme Programming (XP) verwendet. Dieser Ansatz unterteilt das Projekt in mehrere Releases bzw. Iterationen. Dabei gibt es in jeder Iteration ein geregeltes Vorgehen. Zunächst werden von den in der Rolle des Kunden sich befindenden Mitgliedern Userstories zusammen mit der Geschäftsleitung erstelllt. Mit diesen Userstories werden die Anforderungen an das Release beschrieben. Anschließend unterteilen die Entwickler diese in Taskcards, die nun die konkreten Aufgaben enthalten. Die Taskcards werden priorisiert und zeitlich abgeschätzt.

Die einzelnen Aufgaben werden von jeweils zwei Entwicklern mittels Pair-Programming bearbeitet. Dabei sollen die Partner gewechselt werden, um an möglichst allen Aufgaben des Projektes beteiligt zu sein. Dadurch erhält man einen Überblick über das gesamte System.

Die Aufgaben werden mit dem Test-First Ansatz bearbeitet. Hier wird zunächst eine Testklasse geschrieben, und erst wenn diese vollkommen funktionsfähig ist, wird die eigentliche Klasse implementiert. Weiterhin soll der implementierte Code refaktorisiert, also durchgehend überarbeitet und getestet werden.

KAPITEL 3

Übersicht über den Abschlussbericht

Alexander Fronk, Jens Schröder

Der vorliegende Bericht gliedert sich wie folgt:

- Teil 2 gibt die Verschriftlichung der Themen wieder, die in der Seminarphase bearbeitet wurden.
- Teil 3 umfasst kapitelweise die vier Entwicklungszyklen, die die Projektgruppe durchlaufen hat. Jedes Kapitel folgt dem in Kapitel 9 vorgestellten Prozess des eXtreme Programming und ist folglich gegliedert in die Beschreibung der User Stories, der verwendeten Systemmetapher, den erarbeiteten Tasks, der implementierten Architektur sowie der Kundenakzeptanztests.
- Teil 4 gibt einen Überblick über das entwickelte Framework und beschreibt seine Verwendung.
- Der Bericht schließt in Teil 5 mit einem Fazit, welches Ablauf und Organisation der Projektgruppe kritisiert und weiterführende Arbeiten skizziert.
- Im Anhang finden sich die verwendeten Konventionen zur Codegestaltung sowie Hinweise über die Lizenz, unter der das erstellte Produkt genutzt werden darf.

TEIL 2

Seminarphase

KAPITEL 4

Modellierung mit UML

Semih Sevinç

4.1 Einleitung

In dieser Ausarbeitung geht es um die Modellierung objektorientierter Software mit UML, welche man in das statische und dynamische UML-Modell einteilen kann. Bei der Erstellung von Software werden meistens nur die altbekannten Klassendiagramme aus dem statischen Modell verwendet, das dynamische Modell hingegen findet weniger Anwendung. Aber genau wie das statische Modell mit den Anwendungsfall- und Klassendiagrammen, hat das dynamische Modell eine wichtige Aufgabe bei der Modellierung. Es soll das Verhalten des Systems mit Hilfe von entsprechenden Diagrammen wiedergeben. In dieser Ausarbeitung wird daher der Fokus auf das dynamische Modell von UML gelegt und nicht näher auf das statische UML-Modell eingegangen. Die allgemeine Syntax und Semantik von UML werden vorausgesetzt, so dass nicht im Detail auf die einzelnen Elemente eines Diagramms eingegangen wird, sondern lediglich auf die für das Diagramm wesentlichen Elemente. Es sei noch zu erwähnen, dass sich diese Arbeit auf die UML Version 1.4 bezieht und als Literatur *UML@Work* (Hitz und Kappel, 2002) verwendet wurde. In Kapitel 4.2 wird nach einer kurzen Motivation zur Modellierung mit UML eine grafische Übersicht der UML-Diagramme dargestellt. Im darauf folgenden Kapitel wird auf das dynamische Modell eingegangen und die entsprechenden Diagramme mit je einem Beispiel vorgestellt. Im letzten Kapitel erfolgt ein Resümee.

4.2 Motivation

Moderne Software wird meist objektorientiert entwickelt. Aber bevor man objektorientierte Software programmiert, ist es sinnvoll, sie zu modellieren. Die Modellierung hat unter anderem den Vorteil, dass man im Vorfeld der Implementierung schon eventuelle Probleme erkennen und beseitigen kann. Zur Modellierung von objektorientierter Software wird meist die Sprache UML verwendet. Sie bietet für die Modellierung der unterschiedlichen Aspekte eines Systems verschiedene Diagrammtypen an. Die Unified Modeling Language (UML) wurde als Standard zur Modellierung durch die Object Management Group (OMG) akzeptiert. Es gibt inzwischen zahlreiche Werkzeuge zur Modellierung mit UML, wie z.B. Rational Rose, Together und mit Hinblick auf die Projektgruppe, EclipseUML der Firma Omondo. Die unterschiedlichen UML-Diagramme können in den verschiedenen Phasen der Softwareentwicklung

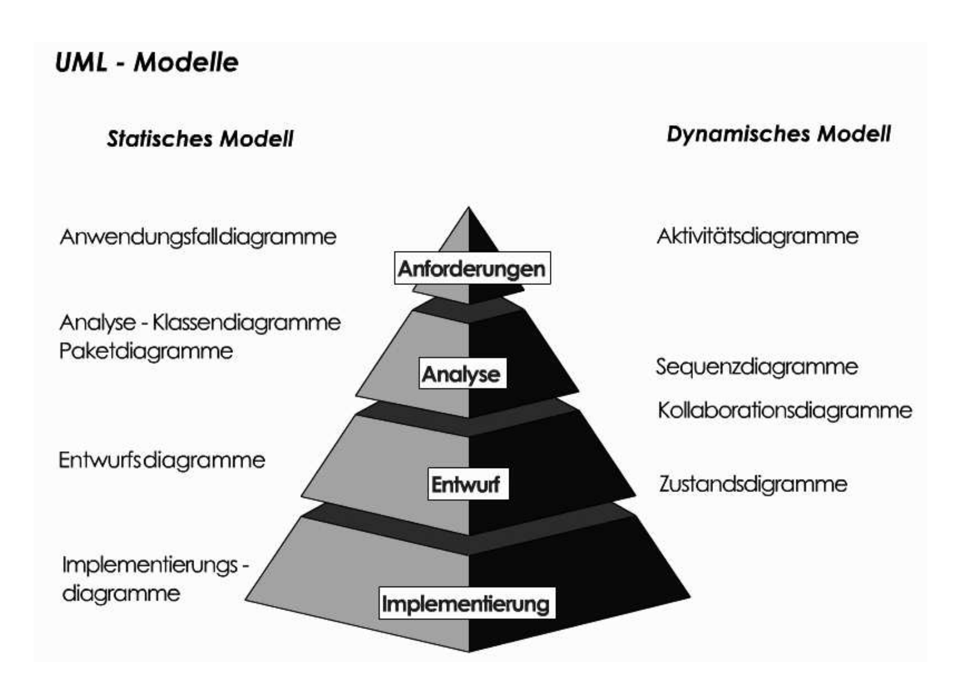


Abbildung 4.1.: UML-Modelle nach W. von Gudenberg

- Anforderungen, Analyse, Entwurf und Implementierung - eingesetzt werden und dem statischen und dem dynamischen UML-Modell zuteilen. Diese Einteilung ist in Abbildung 4.1 dargestellt.

4.3 Das dynamische UML-Modell

Das dynamische Modell besteht aus

- Aktivitätsdiagrammen,
- Sequenzdiagrammen,
- Kollaborationsdiagrammen und
- Zustandsdiagrammen.

Sequenz- und Kollaborationsdiagramme werden auch unter dem Oberbegriff *Interaktionsdiagramme* zusammengefasst. Im nächsten Schritt werden diese UML-Diagramme für die dynamische Modellierung vorgestellt.

4.3.1 Aktivitätsdiagramm

Wie man in Abbildung 4.1 sehen kann, werden Aktivitätsdiagramme meist relativ früh im Entwicklungsprozess eingesetzt, nämlich bei der Anforderungsbeschreibung, wo noch unklar ist, welche Objekte welche Verantwortlichkeit übernehmen. Man erhält einen Überblick über die Aktionen in einem Anwendungsfall und deren Abhängigkeit von weiteren Aktivitäten. Dadurch gewinnt man ein grobes Verständnis für Abläufe des zu modellierenden Systems.

Neben der Modellierung von sequentiellen Abläufen erlauben Aktivitätsdiagramme, bedingte oder parallele Abläufe zu beschreiben. Dadurch werden unnötige Reihenfolgebedingungen vermieden und man kann evtl. Parallelisierungen einbauen, was die Durchlaufzeit des zu modellierenden Geschäftsvorgangs verbessern kann. Zusammengehörende Aktivitäten können in einer so genannten Oberaktivität zusammengefasst werden.

Um in einem Aktivitätsdiagramm deutlich zu machen, welche Aktivität von welcher Rolle ausgeführt wird, kann man Verantwortungsbereiche einzeichnen. Dabei wird das Aktivitätsdiagramm durch vertikale Linien in Bereiche eingeteilt, wobei jeder Bereich eine Verantwortlichkeit darstellt.

Im Hinblick auf die Projektgruppe könnte man mit Hilfe von Aktivitätsdiagrammen die Reihenfolge von Aktivitäten eines Editors verdeutlichen, die ausgeführt werden müssen, um einzelne Anwendungsszenarien, wie etwa das Abspeichern einer Datei auszuführen (siehe Abbildung 4.2): Beim Schließen des Editors hat der Benutzer die Möglichkeit, die Datei zu speichern. Beim Nichtspeichern wird das Programm sofort beendet. Andernfalls erfolgt die Oberaktivität „Datei Speichern“ (graue Box), die wiederum die Aktivitäten „Dateiname eingeben“ und „Datei überschreiben“ beinhaltet. Der Benutzer wird aufgefordert einen Dateinamen einzugeben. Falls dieser Name bereits vorhanden ist, kann er die Datei unter einem anderen Namen speichern oder die bereits vorhandene Datei vom Editor überschreiben lassen. Während die Datei gespeichert wird, sichert ein Controller parallel dazu die Editoreigenschaften, wie z.B. die Ansichteigenschaften, und das Programm wird beendet.

4.3.2 Sequenzdiagramm

In der Analyse- und Entwurfsphase (siehe Abb. 4.1) kommen die Interaktionsdiagrammtypen Sequenz- und Kollaborationsdiagramme zum Einsatz. Ein Sequenzdiagramm modelliert dabei den konkreten Ablauf eines Anwendungsszenarios unter Einbeziehung der beteiligten Objekte. Durch die Betonung auf den zeitlichen Ablauf, wird der Nachrichtenaustausch der Objekte leicht ersichtlich. Den Objekten werden Lebenslinien zugeordnet und der zeitliche Verlauf der Nachrichten wird entlang dieser Lebenslinie modelliert. Die einzelnen Nachrichten werden als waagerechte Pfeile zwischen den Lebenslinien gezeichnet. Auf ihnen wird die Nachricht notiert. Die Antwort auf eine Nachricht wird als gestrichelter Pfeil mit offener Pfeilspitze dargestellt. Die Zeit in der ein Objekt aktiv ist, wird im Sequenzdiagramm durch ein schmales Rechteck, auch *Aktivierungsbalken* genannt, entlang der Lebenslinie dargestellt. Ein Kreuz am Ende des Aktivierungsbalkens symbolisiert die Löschung eines Objektes.

In Abbildung 4.3 ist ein Sequenzdiagramm für das Selektieren eines Objekts dargestellt. Durch das Hineinklicken des Benutzers mit der Maus in die Zeichenfläche gibt es zwei Fallunterscheidungen:

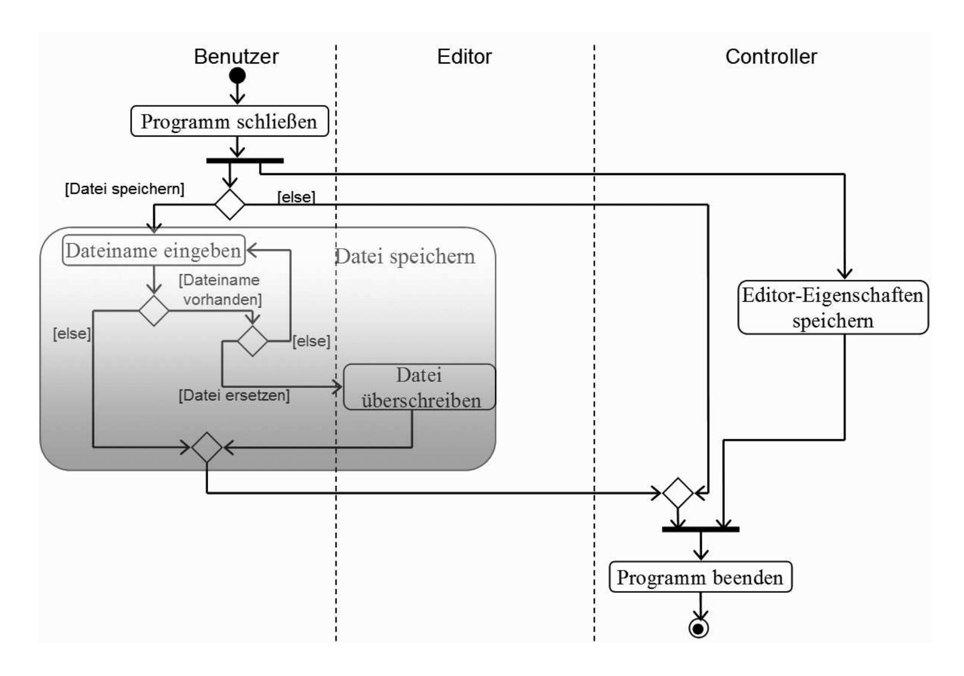


Abbildung 4.2.: Aktivitätsdiagramm zur Abspeicherung einer Datei

1. Der Benutzer trifft eine Figur
2. Der Benutzer erhält als Antwort, dass keine Figur selektiert ist.

Wenn eine Figur selektiert wurde, wird diese der Steuerung übergeben. Diese setzt den Status der Figur als selektiert und aktualisiert ihre Ansicht. Der Benutzer erhält dann die Antwort, dass eine Figur selektiert ist und hätte erst jetzt die Möglichkeit die Figur zu editieren.

4.3.3 Kollaborationsdiagramm

Die zweite Form von Interaktionsdiagrammen stellen die Kollaborationsdiagramme dar. Sie werden genau wie Sequenzdiagramme, in der Analyse- und Entwurfsphase eingesetzt. Ein Kollaborationsdiagramm zeigt im Grunde die gleichen Sachverhalte wie ein Sequenzdiagramm, jedoch aus einer anderen Perspektive: Beim Kollaborationsdiagramm stehen die Objekte und ihre Zusammenarbeit (Kollaboration) untereinander im Vordergrund.

Der zeitliche Verlauf der Kommunikation zwischen den Objekten, der beim Sequenzdiagramm im Vordergrund steht, wird beim Kollaborationsdiagramm durch Nummerierung der Nachrichten verdeutlicht. Dadurch ist leider die Abfolge der Nachrichten nicht mehr so leicht ersichtlich wie im Sequenzdiagramm. Aber dafür hat man mehr Freiheit bei der Anordnung der Objekte, wodurch man Objekte mit intensiven Verbindungen nahe beieinander platzieren kann. Somit kann die Struktur betont und die Lesbarkeit verbessert werden. Genau wie Sequenzdiagramme sind auch Kollaborationsdiagramme geeignet, einzelne Ablaufvarianten mit der Intention zu beschreiben, die Zusammenarbeit mehrerer Objekte in einem Anwendungs-

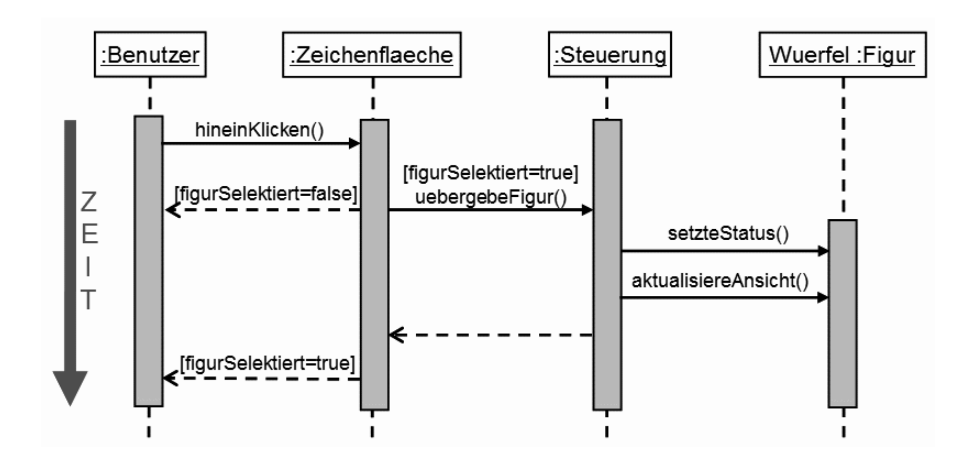


Abbildung 4.3.: Sequenzdiagramm für das Selektieren eines Objekts

fall darzustellen. Sie sind jedoch nicht dazu geeignet, ein Verhalten präzise oder vollständig zu definieren. Hierzu sind Zustandsdiagramme die bessere Wahl.

Ein Kollaborationsdiagramm für das Verschieben eines Objektes wird in der Abbildung 4.4 dargestellt. Damit ein Benutzer einen Würfel verschieben kann, löst er vorher mehrere Mouse-Events aus. In diesem Falle ein Event zum Selektieren und ein weiteres Event zum Verschieben des Würfels. Der Würfel übergibt die neuen Koordinaten der Steuerung, welche die Werte überprüft. Falls die neuen Koordinaten im erlaubten Bereich sind, wird der Würfel entsprechend den neuen Koordinaten verschoben und die Ansicht wird aktualisiert. Durch die Nummerierung wird der zeitliche Ablauf der Nachrichten dargestellt: Die Nachricht „setzeKoordinaten()“ kann erst nach der Nachricht „prüfeKoordinaten()“ erfolgen, welche wiederum von der Nachricht „uebergebeKoordinaten()“ ausgelöst wird.

4.3.4 Zustandsdiagramm

Zustandsdiagramme werden in der Entwurfsphase einer Softwareentwicklung eingesetzt. Es lässt sich mit diesen Diagrammen das Verhalten eines Objektes über mehrere Anwendungsfälle darstellen.

Ein Zustandsdiagramm kann man als einen Graphen mit Zuständen als Knoten und Transitionen als gerichtete Kanten deuten. Die Semantik und Syntax ähnelt denen von Automaten. Es existiert immer ein Start- und ein Endzustand. Eine Transition ist der Übergang von einem Zustand in einen Folgezustand. An diesem Übergang kann ein Ereignis optional mit einer Bedingung geknüpft sein. Sie wird als Pfeil vom Ausgangs- zum Zielzustand dargestellt, der mit dem auslösenden Ereignis beschriftet werden kann. Genau wie bei Aktivitätsdiagrammen, kann man auch hier Zustände zu einem Oberzustand zusammenfassen, wodurch die Lesbarkeit verbessert und die Komplexität verringert wird. Ein weiterer Vorteil von Zustandsdiagrammen ist die so genannte *Nebenläufigkeit*. Wenn beispielsweise ein Objekt verschiedene und unabhängige Verhalten aufweist, dann kann man dies mit nebenläufigen Zustandsdiagrammen

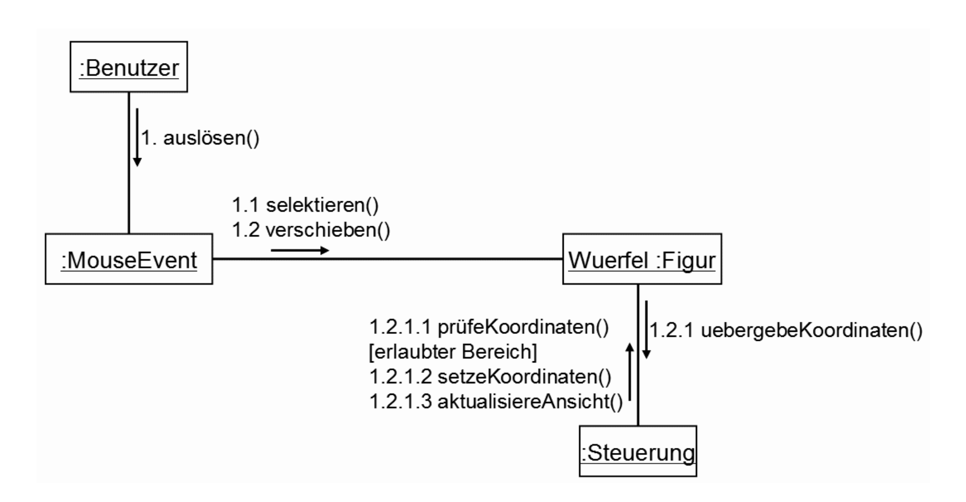


Abbildung 4.4.: Kollaborationsdiagramm für das Verschieben eines Objekts

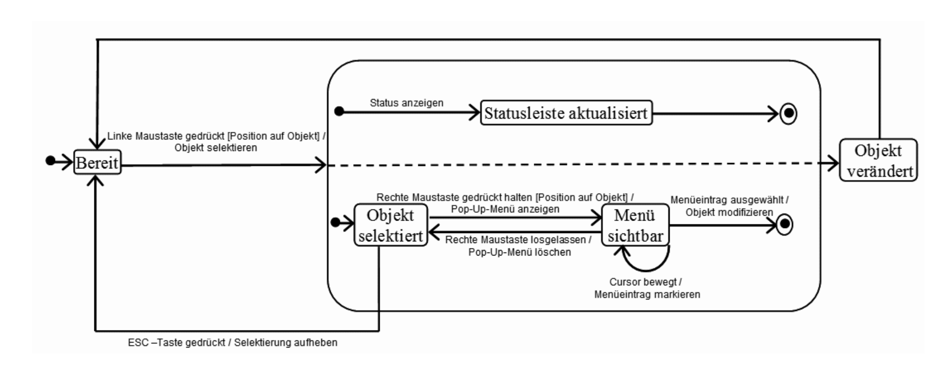


Abbildung 4.5.: Zustandsdiagramm für das Editieren von Objekteigenschaften

besser darstellen.

Für die Projektgruppe wäre der Einsatz dieses Diagrammtyps bei der Darstellung einiger Zustände eines Editors sinnvoll, wie z.B. die Zustände für die Änderung der Eigenschaften eines Objektes über ein Pop-Up-Menü, dargestellt in Abbildung 4.5: Damit ein Pop-Up-Menü für ein Objekt sichtbar ist, muss dieses Objekt vorher mit der linken Maustaste selektiert sein. Hiernach wird die Statusleiste aktualisiert. Parallel dazu tritt beim Ereignis „Rechte Maustaste gedrückt“ mit der Bedingung, dass die Position der Maus auf dem selektierten Objekt ist, die Aktionsfolge „Pop-Up-Menü anzeigen“ ein und das Menü wird sichtbar. Beim Loslassen der rechten Maustaste wird dieses Menü gelöscht und das Objekt ist immer noch selektiert. Mit der ESC-Taste wird die Markierung des Objektes aufgehoben und der Editor befindet sich im Zustand „Bereit“. Während das Menü sichtbar ist, wird bei jeder Cursorbewegung der Menüeintrag markiert. Nachdem man einen Menüeintrag ausgewählt hat, wird das Objekt dementsprechend geändert und der Editor befindet sich erneut im Startzustand „Bereit“.

4.4 Zusammenfassung

Die einzelnen Diagramme des dynamischen Modells begleiten die Softwareentwicklung während des Entwicklungsprozesses. Durch Aktivitätsdiagramme kann man am Anfang der Softwareentwicklung schon ein grobes Verständnis für die Abläufe des zu modellierenden Systems haben.

In der Analyse- und Entwurfsphase verdeutlichen Interaktionsdiagramme das ablauforientierte Verhalten von Operationen. Dabei machen Sequenzdiagramme zeitliche Abläufe auf einem Blick deutlich. Jedoch leidet die Übersichtlichkeit, wenn viele Objekte mit einem hohen Nachrichtenaustausch dargestellt werden. Dieser Nachteil kann mit Kollaborationsdiagrammen durch die freie Anordnung der Objekte verringert werden, um so strukturelle Zusammenhänge zu verdeutlichen. Ihr Nachteil gegenüber Sequenzdiagrammen besteht darin, dass zeitliche Abläufe nicht sofort erfassbar sind. Leider geht die Einfachheit und Klarheit der Interaktionsdiagramme rasch verloren, wenn man komplizierte Prozesse mit vielen Schleifen und Fallunterscheidungen hat.

Zustandsdiagramme beschreiben erlaubte Aufrufreihenfolgen für die Operationen auf einem Objekt und modellieren somit den Lebenszyklus eines Objektes.

Mit all diesen Diagrammen des dynamischen Modells kann das Verhalten eines Systems gut wiedergeben werden. Durch die Kombination der Interaktionsdiagramme kann man die Nachteile eines Diagramms, durch die Vorteile des anderen kompensieren. Die dynamische Modellierung stellt zusammen mit dem statischen Modell eine wichtige und hilfreiche Rolle bei der Entwicklung von Software dar.

KAPITEL 5

Constraint Multiset Grammars

Stephan Eisermann

5.1 Einleitung

Constaint Multiset Grammars (CMG) definieren die Syntax von visuellen Sprachen. Visuelle Sprachen werden zur Beschreibung und Erkennung von graphischen Eingaben wie beispielsweise Diagrammen eingesetzt.

In dieser Arbeit wird eine informelle Einführung der Syntax zur Beschreibung von textuellen Sprachen, der *erweiterten Backus-Naur Form (EBNF)* und ihre Abgrenzung gegenüber der CMG gegeben. Im Anschluß hieran folgt die formale Definition der Constranint Multiset Grammars. Ein Beispiel soll den Einsatz der CMG verdeutlichen. Abschließend werden die Komplexität verschiedener Klassen von Constraint Multiset Grammars und eine mögliche algorithmische Erkennung behandelt.

5.2 Grammatiken

5.2.1 Abgrenzung EBNF und CMG

Constraint Multiset Grammars sind ein Ansatz zur Definition einer visuellen Sprache. Die EBNF ist hingegen eine Grammatik, die zur Beschreibung von Programmiersprachen verwendet wird. Visuelle Sprachen unterscheiden sich in einem wichtigen Punkt von textuellen Sprachen: In textuellen Sprachen erfolgt die Eingabe von Zeichen von links nach rechts, entsprechend erfolgt auch die Erkennung durch einen Parser. Allerdings gibt es keine natürliche Reihenfolge in der beispielsweise ein Zustandsdiagramm gezeichnet werden muss (Helm und Marriott, 1991), daher gibt es auch keine Reihenfolge in der der Parser die Zeichen bearbeiten muss.

Der Aufbau von beiden Grammatiken ist ähnlich, wobei die EBNF, die hier informell an einem Beispiel eingeführt wird, einen einfacheren Aufbau besitzt. Man unterscheidet in der EBNF *terminale* Zeichen und *nicht-terminale* Zeichen. Nicht-terminale Zeichen können durch eine Produktion durch eine Sequenz von terminalen und nicht-terminalen Zeichen ersetzt werden. Eine Produktion hat ein nicht-terminales Zeichen auf der linken Seite, auf der rechten Seite eine Sequenz von terminalen und nicht-terminalen Zeichen. Eine Binärziffer könnte wie folgt beschrieben werden:

```
Binärziffer::="0 "1".
```

Diese einfache Produktion sagt aus, dass das nicht-terminale Zeichen `Binärziffer` durch die beiden terminalen Zeichen `0` oder `1` ersetzt werden kann, wobei „::=“ die Bedeutung von *wird zu* hat und „|“ die eines *oder-Operators*. Eine Binärziffer kann also 0 oder 1 sein.

Bis hier ist die Definition der Grammatiken gleich. Es folgt die formale Definition der CMG, die auch auf die Unterschiede eingeht.

5.2.2 Formale Definition der CMG

Zeichen

Grafische Zeichen sind verglichen mit textuellen Zeichen viel komplexer. Um alle Informationen zu einem (grafischen) Zeichen aufzunehmen, benötigt man eine Liste von Attributen. Dieses wird klar, wenn man sich beispielsweise die Beschriftung eines Pfeiles in einem Zustandsübergangsdiagramm anschaut. Es gibt Attribute, die eher geometrische Informationen darstellen (Mittelpunkt, Höhe) und Attribute, die semantische Informationen darstellen (Typ der Beschriftung, z.B. *string* oder *int*) . Entsprechend lassen sich auch die Attribute von Zeichen im allgemeinen grob in zwei Gruppen einteilen, nämlich in die Gruppe, die geometrische Informationen darstellt, und in die Gruppe, die semantische Informationen darstellt. Zeichen in der CMG können wie in der EBNF terminal oder nicht-terminal sein, wobei nicht-terminale Zeichen durch eine Produktion durch eine Menge von terminalen und nicht-terminalen Zeichen ersetzt werden können. Marriot definiert ein Zeichen dann wie folgt (Marriott, 1994):

Definition 5.2.1
Ein Zeichen $T(\vec{\Theta})$ besteht aus einem Typen T und einer Folge von Elementen aus einer Liste von Attributen (computation domain), die dem Zeichen zugeordnet sind, $\vec{\Theta}$, die eine Zuweisung der Attribute von T darstellen. T kann ein terminaler Typ, ein nicht-terminaler Typ oder ein Starttyp sein, und das Zeichen wird dann entsprechend terminales Zeichen, nicht-terminales Zeichen oder Startzeichen genannt.

Constraint Multiset Grammar

Die folgende Definition von Constraint Multiset Grammars stammt ebenfalls von Marriott (Marriott, 1994):

Definition 5.2.2
Eine CMG in einer computation domain D besteht aus

- *einer Menge von Zeichen T_T, deren Typ terminal ist;*
- *einer Menge von Zeichen T_{NT}, deren Typ nicht-terminal ist;*
- *einem ausgezeichnetem Zeichen $S_T \in T_{NT}$, das vom Typ Start ist;*
- *einer Menge von Produktionen.*

Jedes Zeichen $t \in T_T \cup T_{NT}$ besitzt eine Liste von Attributen. Das Startzeichen darf nur auf der linken Seite einer Produktion auftauchen. Produktionen haben die Form:

$$T(\vec{x}) ::= T_1(\vec{x_1}), \ldots, T_n(\vec{x_n}) \quad where \quad exists \quad T'_1, \ldots, T'_m \quad where \quad C \quad \vec{x} = F$$

wobei gilt

- *T ist ein nicht-terminales Zeichen;*
- *$T_1, \ldots, T_n$ sind Zeichen eines Typs mit $n \geq 1$;*
- *$T'_1, \ldots, T'_m$ sind Zeichen eines Typs mit $m \geq 0$;*
- *$\vec{x}, \vec{x_i}, \vec{x_i}'$ sind Listen von Variablen;*
- *C ist eine Verknüpfung von Bedingungen über $\vec{x_1}, \ldots, \vec{x_n}, \vec{x_1}', \ldots, \vec{x'_m}$*
- *F ist eine Funktion von $\vec{x_1}, \ldots, \vec{x_n}, \vec{x'_1}, \ldots, \vec{x'_m}$*

Auch hier werden wieder Unterschiede zur EBNF sichtbar. Damit die Produktion ausgeführt werden kann, muss die Bedingung C erfüllt sein. Diese Einschränkung ergibt sich aus der schon in 2.1 angesprochenen Tatsache, dass die Sequenz als Bedingung für grafische Eingaben nicht von Bedeutung ist. Eine weitere Besonderheit ist, dass optional die Existenz von bestimmten Zeichen vorausgesetzt werden kann. Auf diese Bedingung wird näher in 2.4 eingegangen. Da die Zeichen im Unterschied zur EBNF Listen von Variablen haben, müssen die Attribute des entstehenden Zeichens auf der linken Seite der Produktion durch eine Funktion aus den Attributen der Zeichen auf der rechte Seite belegt werden.

5.2.3 Beispiel

Die Beschreibung des Zustandsübergangsdiagramms in Abbildung 1 soll den Einsatz der Constraint Multiset Grammar verdeutlichen. Ziel ist es, den Startzustand zu beschreiben, wobei hierzu die Definition der CMG aus 2.2 herangezogen wird.

Ein Zustand wird in einem Zustandsübergangsdiagramm durch einen Kreis beschrieben, der den Namen des Zustandes einschließt. Die folgende Produktion beschreibt einen Zustand:

$$
\begin{aligned}
&state(P_{midpoint}, P_{radius}, P_{name}, P_{kind}) ::= \\
&\quad circle(Q_{midpoint}, Q_{radius}), text(T_{midpoint}, T_{height}, T_{width}, T_{string}) \\
&\quad where \\
&\qquad Q_{midpoint} = T_{midpoint}, \\
&\qquad 2 * Q_{radius} \geq T_{height}, \\
&\qquad 2 * Q_{radius} \geq T_{width} \\
&\quad and \\
&\qquad P_{midpoint} = Q_{midpoint}, \\
&\qquad P_{radius} = Q_{radius}, \\
&\qquad P_{name} = T_{string},
\end{aligned}
$$

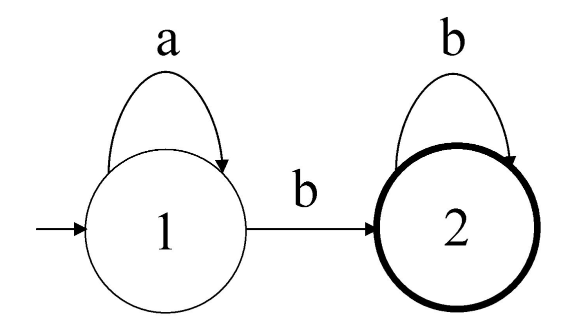


Abbildung 5.1.: Beispiel für ein Zustandsübergangsdiagramm

$P_{kind} = normal.$

Auf der linken Seite der Produktion steht ein Zeichen des Typs *state*, das die Attribute *midpoint*, *radius*, *name* und *kind* hat. Dieses kann aus den beiden Zeichen vom Typ *circle* und vom Typ *text* entstehen. Die Produktion kann aber nur angewendet werden, wenn die Zeichen die Bedingungen erfüllen, die durch die *where-Klausel* aufgeführt werden. In diesem Beispiel müssen die Mittelpunkte der beiden Zeichen *circle* c, und *text* t, gleich sein, der Durchmesser des Zeichens c muss größer oder gleich der Höhe des Zeiches t sein, gleiches gilt für die Breite des Zeichens t. Sind diese drei Bedingungen erfüllt, können c und t zu einem Zeichen s vom Typ *state* zusammengefasst werden. Der Mittelpunkt von s entspricht dann dem Mittelpunkt von c, der Radius von s entspricht dem Radius von c und der Name von s dem von t. Die Belegung des Attributes *kind* mit *normal* zeigt an, dass es sich um einen „einfachen" Zustand handelt, also nicht um einen Endzustand.

5.2.4 Bedingungen

Bedingungen nehmen bei der Spezifikation für eine visuelle Sprache eine Schlüsselstellung ein. Sie ermöglichen es, Informationen über das räumliche Layout und Beziehungen zwischen einzelnen Elementen *direkt* in der Grammatik zu codieren.

Des Weiteren werden sie genutzt, um zu überprüfen, ob eine Produktion angewendet werden kann oder nicht. Sie definieren die Attribute der Zeichen auf der rechten Seite der Produktion durch Ausdrücke der Attribute von Zeichen auf der linken Seite (siehe auch Kapitel 2.3).

Negative Bedingungen sind einer der Hauptunterschiede zwischen CMGs und anderen Formalismen, mit denen visuelle Sprachen definiert werden. Ohne die Verneinung wäre es schwierig viele existierende visuelle Sprachen, z.B. Zustandsübergangsdiagramme oder binäre Bäume, mit einer deterministischen Grammatik zu beschreiben. Ohne deterministische Grammatik ist es schwierig, einen Parser zu bauen, der eine auf dieser Grammatik basierende Sprache effizient erkennen soll (siehe auch Kapitel 3). Hierbei werden die folgenden Arten von Bedingungen unterschieden:

Topological constraints erlauben es, mit Hilfe von Tests über die räumliche Anordnung von unterschiedlichen Zeichen (z.B. A enthält B) zu überprüfen, ob eine Produktion durchgeführt werden soll oder nicht.

Eine beispielhafte Produktion hierfür beschreibt einen Endzustand aus dem Diagramm in Abbildung 1. Der Endzustand muss aus zwei Kreisen bestehen, wobei ein Kreis in dem anderen enthalten sein muss (*contains*) und der innere Kreis ein Textfeld enthalten muss.

$state(P_{area}, P_{name}, P_{kind}) ::=$
$circle(Q_{area}), circle(R_{area}), text(T_{area}, T_{string})$
$where$
$Q_{area}\ contains\ R_{area},$
$R_{radius}\ contains\ T_{area}$
and
$P_{area} = Q_{area},$
$P_{name} = T_{string},$
$P_{kind} = final.$

Minimization constraints werden hauptsächlich dazu genutzt, das beste (beispielsweise das in der unmittelbaren Nähe befindliche) Objekt auszuwählen, das die Bedingungen erfüllt.

Als Beispiel dient hier die Beschreibung eines Pfeiles mit einer Beschriftung (Abbildung 1). Dabei muss das Textfeld die Entfernung seines Mittelpunktes zum Mittelpunkt des Pfeiles minimieren und seine *area* muss gleichzeitig über dem Mittelpunkt des Pfeiles liegen.

$arc(P_{start}, P_{end}, P_{label}) ::=$
$arrow(Q_{start}, Q_{midpoint}, Q_{end}), text(T_{area}, T_{string})$
$where$
$(T\ minimizes\ distance(T_{center}, Q_{midpoint})$
$where$
$T_{area}\ above\ Q_{midpoint})$
and
$P_{start} = Q_{start},$
$P_{end} = Q_{end},$
$P_{label} = T_{text}.$

Existental quantification wird genutzt um zu testen, ob bestimmte Zeichen existieren. Existieren diese Zeichen nicht, so kann die Produktion auch nicht ausgeführt werden.

Diese folgende Produktion erkennt eine Transition zwischen zwei Zuständen (Abbildung 1). Damit diese Transition erkannt werden kann, müssen natürlich zwei Zustände existieren, die von einem Pfeil als Start- und Endzustand berührt werden.

$tran(T_{from}, T_{to}, T_{label}) ::=$
$arc(A_{start}, A_{end}, A_{label})$
$where\ exists$
$state(R_{area}, R_{kind}),$
$state(S_{area}, S_{kind})$
$where$
$A_{start}\ touches\ R_{area},$
$A_{end}\ touches\ S_{area}$
and

$T_{from} = R_{name},$
$T_{to} = S_{name},$
$T_{label} = A_{label}.$

Negativ constraints werden genutzt, um Produktionen zu definieren, die davon abhängen, dass bestimmte Zeichen nicht existieren.

Diese Bedingung verlangt, dass in dem Zeichen von Typ *box* nur genau ein anderes Zeichen von Typ *picture* liegen darf.

$box(B_{dimension})$ *containsOnly* $picture(P_{dimension})$
if
B contains P,
not exists
$picture(Q_{dimension})$
where
$Q <> P$,
B contains Q

5.3 Komplexität

Das *Membership-Problem* beschreibt das Problem festzustellen, ob eine bestimmte Familie von Zeichen in einer bestimmten Sprache enthalten ist oder nicht. Voraussetzung ist, dass das Membership-Problem entscheidbar ist, um einen entsprechenden Algorithmus zu entwickeln. Die Entscheidbarkeit ist für eine beliebige CMGs nicht geben (Marriott, 1994).

Das Problem lässt sich nur für kreisfreie CMGs entscheiden. Eine CMG wird genau dann *kreisfrei* genannt, wenn keine Produktion existiert, die ein nicht-terminales Zeichen in ein anderes nicht-terminales Zeichen umformen kann. Allerdings ist das Membership-Problem für kreisfreie CMGs immer noch NP-hart.

5.3.1 Inkrementelles Parsen

Um das Membership-Problem für kreisfreie CMGs zu lösen, hat Marriot in (Marriott, 1994) einen effizienten inkrementellen bottom-up Parsing-Algorithmus entwickelt. Diesem fehlte allerdings noch die Möglichkeit, mit negativen Bedingungen zu arbeiten. In (Chok und Marriott, 1995) wurde dieser Algorithmus entsprechend weiterentwickelt.

Hier soll kurz die Arbeitsweise eines einfachen Algorithmus, der nur auf Grammatiken mit positiven Bedingungen arbeitet, vorgestellt werden. Der Algorithmus startet mit einer Menge von terminalen Zeichen. Auf dieser Menge werden wiederholt Produktionen ausgeführt, was dazu führt, dass terminale Zeichen zu nicht-terminalen Zeichen zusammengefasst werden und *Parsebäume* entstehen. Ein Parsebaum ist ein Baum von Zeichen, in dem jedes Blatt eine assoziierte Produktion hat. Die wiederholte Anwendung von Produktionen führt zu immer größer werdenden Parsebäumen. Das Ziel ist es, nur noch einen Parsebaum zu haben, dessen Wurzel ein nicht-terminales Zeichen von Typ *start* ist. Ein Wald von Parsebäumen wiederum ist eine Datenstruktur, die eine Menge von Parsebäumen enthält. Der Algorithmus terminiert, sobald sich der Wald nicht mehr durch Anwendung von Produktionen verändern lässt.

Um die Effizienz des Algorithmus zu steigern, dürfen nicht alle Produktionen auf einmal betrachtet werden. Es wird ein so genannter *call graph* erzeugt, der die Abhängigkeiten zwischen Produktionen in der Grammatik enthält. Eine Produktionsregel P1 ist von einer anderen Produktionsregel P2 abhängig, wenn gilt: Ein Zeichen auf der rechten Seite von P1 hat den gleichen Typen wie ein Zeichen auf der linken Seite von P2. Es werden die stark zusammenhängenden Komponenten (SZK) berechnet und geordnet. Produktionsregeln in höheren SZKs hängen von Regeln in tieferen SZKs ab. Der Algorithmus versucht, zuerst die Regel in der untersten SZK anzuwenden und bewegt sich eine Ebene höher, falls keine Produktionsregel mehr anwendbar ist.

5.4 Zusammenfassung

Es wurde gezeigt, dass zur Beschreibung von visuellen Sprachen Constraint Multiset Grammars eingesetzt werden können. Die formale Definition der CMGs wurde vorgestellt und in einem Beispiel erläutert. Die Komplexität eines Parsers für CMGs wurde angesprochen, danach ein einfacher Algorithmus vorgestellt, der zum Bau von Parsern benötigt wird. Abschließend lässt sich sagen, dass CMGs für die Definition von visuellen Sprachen gut geeignet sind und auch der Bau eines Parsers als Grundlage für einen Editor möglich, aber offensichtlich schwierig und ineffizient ist.

Kapitel 6

Graphgrammatiken

Armin Bruckhoff

6.1 Einleitung

Das Ziel der Projektgruppe ist es, ein Editor-Framework zu schaffen, in dem in einer dreidimensionalen Darstellung Software-Strukturen erstellt und verändert werden können. Die durch Manipulation durch den Anwender veränderte Darstellung muss natürlich korrekte Softwarestrukturen enthalten. In einem Java-Klassendiagramm zum Beispiel muss stets gewährleistet sein, dass eine Klasse nur eine Superklasse hat, oder dass es keine zyklischen Vererbungen gibt.

Grammatiken sind ein geeignetes Mittel, bestimmte Strukturen in Zeichenketten zu prüfen. Aber auch Grammatiken, die Graphen auf syntaktische Korrektheit überprüfen, werden schon über einen langen Zeitraum erforscht.

Die in einem Editor dargestellten Software-Strukturen können intern durch Graphen repräsentiert werden. Der Typ des Graphen muss allerdings erst noch genau definiert werden und eine Graphgrammatik für diesen Typ aufgestellt werden. Mit dieser Graphgrammatik kann der Editor dann überprüfen, ob der Graph nach der Manipulation durch den Anwender noch einen korrekten Graphen darstellt.

Im folgenden wird zunächst eine kurze Einführung zu Grammatiken für Zeichenketten und für Graphen gegeben. Anschließend wird mit dem Einbettungsproblem ein Problem vorgestellt, dem sich Graphgrammatiken zu stellen haben, und drei Lösungsmöglichkeiten dafür beschrieben. Danach folgt eine Übersicht über verschiedene Ansätze, Graphgrammatiken formal zu beschreiben. Anhand des dritten vorgestellten Ansatzes wird abschließend noch das Konzept des Parsens beschrieben und ein Algorithmus zur Entscheidung des Sprachproblems vorgestellt.

6.2 Einführung zu Grammatiken

6.2.1 Grammatiken für Zeichenketten

Grammatiken beschreiben Verfahren, mit denen formale Sprachen erzeugt bzw. erkannt werden können (Rechenberg, 1999). Eine formale Sprache L ist eine Menge von Zeichenketten, die aus einem Alphabet V gebildet werden: $L \subseteq V^*$, wobei V^* die Menge aller Zeichenketten

inklusive der leeren Zeichenkette ist, die aus dem Alphabet gebildet werden können. $(V^*, \circ)$ ist der freie Monoid über V, und $\circ$ die assoziative Verknüpfung in V. Die Elemente der Menge L werden meist als Wörter der Sprache bezeichnet.

Eine Grammatik ist nun eine Sammlung von Produktionsregeln, die eine Zeichenkette in eine andere überführt. Diese Produktionen enthalten zwei verschiedene Arten von Zeichen. Zum einen gibt es Terminalzeichen, die das Alphabet der Sprache bilden, zum anderen Nichtterminalzeichen, die von Produktionen mit Terminal- und Nichtterminalzeichen ersetzt werden. Erst, wenn keine Nichtterminalzeichen in der Zeichenkette vorhanden sind, kann diese ein Wort der Sprache sein.

6.2.2 Graphgrammatiken

Graphgrammatiken beruhen auf den gleichen Prinzipien und Vorgehensweisen wie Grammatiken für Zeichenketten. Allerdings werden sie, wie der Name schon anzeigt, in Form von Graphtransformationen notiert und sind daher für die Konstruktion von von Graphen geeignet. Mittels einer Graphgrammatik kann überprüft werden, ob ein gegebener Graph zu einer Klasse von Graphen gehört. So gibt es für ER-Diagramme ebenso eine Grammatik wie für Process-Flow-Diagramme oder Abstrakte Syntax Graphen (Rekers und Schürr, 1997).

Graphgrammatiken beinhalten analog zu Grammatiken für Zeichenketten eine Sammlung von Regeln, die beschreiben, wie die bestehende Struktur in eine neue Struktur überführt werden kann, sowie eine Menge von Nichtterminalzeichen und eine Menge von Terminalzeichen. Zeichen sind allerdings bei Graphen nicht einfache Buchstaben eines Alphabets, sondern Knoten und Kanten. Hier kann es je nach Typ des Graphen verschiedene Arten von Knoten geben. ER-Diagramme zum Beispiel besitzen drei verschiedene Knotenarten: Entitäten, Relationen und Attribute. Nichtterminalzeichen sind auch hier nur in den Zwischenrepräsentationen eines Graphen während der Erzeugung oder Erkennung vorhanden. Das Aussehen der Nichtterminaleichen spielt dabei keine Rolle, sie sind nur für die Grammatik nötig und können in der Implementierung frei belegt werden.

Bei Produktionen spricht man meist von linken und rechten Seiten. Die linke Seite enthält die Ausgangssituation vor dem Anwenden, die rechte Seite entsprechend das Ergebnis der Produktion. Produktionen können nacheinander in beliebiger Reihenfolge und beliebig oft auf einen Graphen angewendet werden, solange im Graph Nichtterminalzeichen vorhanden sind, die der linken Seite einer Produktion entsprechen. Begonnen wird dabei immer mit dem Axiom. Das Axiom ist eine spezielle Produktion, bei der aus dem leeren Graph der initiale Startgraph generiert wird. Der leere Graph ist dabei analog zum leeren Wort der Grammatiken für Zeichenketten definiert.

Durch Anwenden der Produktionen entstehen dann die Graphen, die diese Grammatik erzeugen kann. Die Menge der erzeugbaren Graphen ist die Sprache der Grammatik. Analog zu Grammatiken für Zeichenketten werden die Graphen als Worte der Sprache bezeichnet.

Ein (willkürliches) Beispiel für eine solche Produktion ist in Abbildung 6.1 zu sehen. Die linke Seite enthält zwei Knoten, die über eine gerichtete Kante verbunden sind. Auf der rechten Seite sind diese beiden Knoten noch immer vorhanden. Die Kante zwischen ihnen ist durch zwei neue Knoten und vier gerichtete Kanten ersetzt worden. Die Richtung der neuen Kanten entspricht der Richtung der vorher vorhandenen Kante. Der Quellknoten im Ausgangsgraph ist auch im neuen Graph der Quellknoten, analog verhält es sich beim Senkenknoten.

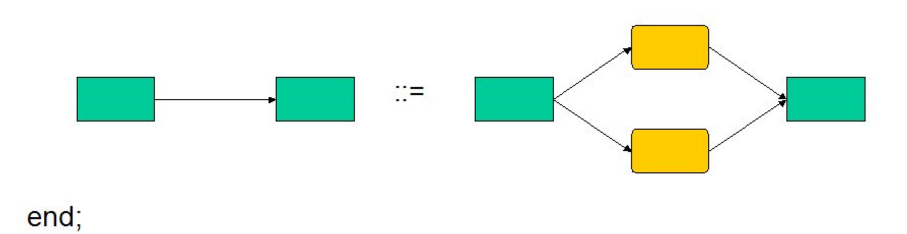


Abbildung 6.1.: Beispiel für eine Produktion

6.2.3 Das Einbettungsproblem bei Graphgrammatiken

Im vorigen Beispiel sind die beiden Knoten der linken Seite auch auf der rechten Seite noch vorhanden. Dies ist aber nicht zwingend gefordert. Es kann also vorkommen, dass Knoten durch eine Produktion gelöscht werden. Dabei muss natürlich beachtet werden, dass der Knoten unter Umständen noch mit Knoten verbunden war, die nicht in der Produktion erfasst worden sind. Dies ist ein besonderes Problem von Graphgrammatiken.

In Grammatiken für Zeichenketten tritt dieses Problem nicht auf. Sie sind linear aufgebaut; es gibt in kontextfreien Grammatiken genau ein Zeichen vor einem Nichtterminalzeichen und genau eins dahinter. In kontextsensitiven Grammatiken kann statt des einzelnen Nichtterminalzeichens auch eine Folge von Terminal- und Nichtterminalzeichen stehen. Diese Folge entspricht einer kompletten linken Seite einer Produktion.

Bei Graphen können die sogenannten Kontextelemente, also Knoten, mit denen ein Nichtterminalzeichen in Beziehung steht, nahezu beliebig um das Nichtterminalzeichen herum angeordnet sein. Es kann also nicht genau gefolgert werden, wo die nach der Produktion neu enstandenen Elemente in diese Beziehungen eingefügt werden müssen. So könnte es z.B. sinnvoll sein, eine Kante, die zu dem ersten Knoten in der Produktion aus Abbildung 6.1 zeigt, nach Anwenden der Produktion auf den oberen neu entstandenen Knoten zeigen zu lassen.

Um das Einbettungsproblem zu lösen, werden drei verschiedene Möglichkeiten häufig genutzt (Rekers und Schürr, 1997, Seite 5). Diese sind das *Erweitern mit Kontextelementen*, das *Implizite Einbetten* und die *Speziellen Einbettungsregeln*. Sie werden im folgenden kurz beschrieben und jeweils ihre Vor- und Nachteile aufgezeigt.

Hinzufügen weiterer Kontextelementen

Hier werden weitere Kontextelemente in die Produktionen mit aufgenommen. Kontextelemente sind Knoten, die mit den direkt von der Produktion betroffenen Knoten unmittelbar in Beziehung stehen. So können Beziehungen, die vorher nicht exakt bestimmbar waren, genau festgelegt und entsprechend gezogen werden.

Ein Vorteil dieses Vorgehens ist es, dass die Produktionen leichter lesbar und verständlicher

werden, da sie einen größeren Ausschnitt aus dem Graphen enthalten. Eine Begrenzung der Beziehungen, die ein Element haben darf, ist im Allgemeinen nicht vorhanden. Das hat für einen Parsingalgorithmus den gravierenden Nachteil, dass er mit Produktionen arbeiten muss, die zum Teil wesentlich mehr Elemente enthalten als Produktionen, die nur die direkt von der Produktion betroffenen Elemente enthält. Der Algorithmus wird somit komplexer und verliert deutlich an Performanz.

Implizites Einbetten

Das Implizite Einbetten findet man bei Constraint Multiset Grammars (Chok und Marriott, 1995) und Picture Layout Grammars (Golin, 1991). Bei diesen Grammatiken wird nicht zwischen Knoten und Kanten unterschieden. Alle Beziehungen der Objekte werden über ihre Attribute und Beschränkungen ihrer Werte definiert. Sie sind also nur implizit vorhanden. Werden nun in Produktionen Attribute neu zugewiesen, so entstehen unter Umständen neue Beziehungen zu Objekten, die nicht im aktuellen Kontext der Produktion enthalten sind. Es muss also genau darauf geachtet werden, den Attributen nur solche Objekte als Werte zuzuweisen, die in der Produktion benutzt werden.

Neben den Seiteneffekten durch solche „falschen“ Zuweisungen hat der Ansatz des impliziten Einbettens den Nachteil, dass die Informationen über Beziehungen nicht leicht erkennbar sind. Ein Parser benötigt viel Zeit und Resourcen, die Beziehungen aus den Attributen aller Objekte auszulesen und zu verarbeiten.

Spezielle Einbettungsregeln

Der dritte Ansatz besteht darin, für die verschiedenen Situationen, in denen Unklarheit über die zu setzenden Beziehungen besteht, verschiedene Regeln anzubieten. Somit gibt es eine wesentlich größere Menge von Produktionen, da für alle möglicherweise auftretenden Sonderfälle eine eigene Produktion vorhanden ist. Daher ist eine Grammatik, die mit diesen speziellen Einbettungsregeln arbeitet, nur schwer zu verstehen.

Auch einzelne Produktionen sind nicht sehr handlich, da immer genau darauf geachtet werden muss, mit welchen Elementen man es zu tun hat, und welche Produktion dann genau betroffen ist. Die gleichen Probleme bestehen natürlich auch für den Parsingalgorithmus. Er muss für jede Produktion den Graphen überprüfen und nach möglichen Anwendungsstellen suchen. Bei der großen Anzahl von Produktionen ist dieser Vorgang sehr aufwendig, so dass alle Algorithmen, die für diesen Ansatz existieren, meist ineffizient sind oder die linken und rechten Seiten der Produktionen zu stark einschränken.

Ein weiterer, sehr gewichtiger Nachteil ist, dass der Ansatz des impliziten Einbettens es nicht ermöglicht, neue Beziehungen zwischen bereits vorhandenen Knoten zu erstellen.

6.3 Beschreibungsansätze

Es gibt in der Literatur verschiedene Ansätze, Graphgrammatiken formal zu erfassen (Schürr und Westfechel, 1992). Dabei sind mehrere Hauptrichtungen auszumachen: der mengentheoretische, der kategorientheoretische Ansatz und der graphentheroetische. Die Erforschung der ersteren begann bereits in den sechziger Jahren. Der graphentheoretische Ansatz nach Rekers

und Schürr ist erst Ende der neunziger Jahre entstanden.

6.3.1 Mengentheoretischer Ansatz

Der mengentheoretische Ansatz hat seinen Namen dadurch erhalten, dass hier die Kanten und Knoten in Mengen verwaltet werden. Operationen, die einen Graphen verändern, sind als Mengenoperationen darstellbar. Alle Produktionen der Grammatik werden so durch Vereinigungen, Durchschnitte, Differenzen etc. beschrieben. Da Mengenoperationen eine intuitive mathematische Grundlage haben, ist der mengentheoretische Ansatz leichter verständlich als der kategorientheoretische.

Der mengentheoretische Ansatz ist allerdings nur in der Lage, kontextfreie Graphgrammatiken zu beschreiben. Bei kontextfreien Grammatiken steht auf der linken Seite der Produktion jeweils nur ein Nichtterminalzeichen, das dann durch mehrere Terminal- oder Nichtterminalzeichen ersetzt wird. Für viele Graphentypen werden dadurch die Produktionen sehr komplex und die Anzahl der Produktionen nimmt zu.

6.3.2 Kategorientheoretischer Ansatz

Der zweite wichtige Ansatz ist der kategorientheoretische, auf den nicht näher eingegangen wird.

Mit diesem Ansatz ist es möglich, auch kontextsensitive Graphgrammatiken zu erfassen. Hier können nun auf beiden Seiten einer Produktion Terminal- und Nichtterminalzeichen stehen. Es können so ganze Teilgraphen von einer Produktion verändert werden.

Produktionen sind leichter aufstellbar, allerdings sind die mathematischen Grundlagen der Kategorien äußerst komplex und erschweren das Verständnis dieses Ansatzes.

6.3.3 Graphentheoretischer Ansatz nach Rekers und Schürr

Zur Beschreibung der Graphgrammatik verwenden Rekers und Schürr Graphen. Die Kandidaten für das Anwenden von Produktionen sind somit im Graph relativ einfach aufzufinden und ihre Auswirkungen leicht verständlich.

Für ihren Ansatz haben Rekers und Schürr eine neue Klasse von Grammatiken, die sogenannten *Layered Graph Grammars* (LGG) definiert. LGGs sind eine Verfeinerung der kontextsensitiven Graphgrammatiken. Auf beiden Seiten der Produktionen dürfen Teilgraphen stehen. Es wird dabei aber gefordert, dass die linke Seite einer Produktion lexikographisch kleiner sein muss als die rechte Seite. Somit ist es nicht mehr möglich, zyklische Ableitungen zu bilden. Eine zyklische Ableitung wäre das Ausführen zweier oder mehr Produktionen nacheinander an der gleichen Stelle des Graphen, das dann wieder zur ursprünglichen Situation führen würde.

Durch das Ausschließen der zyklischen Ableitungen wird das Problem vermieden, Produktionen von kontextsensitiven Graphgrammatiken unkontrollierbar oft ausführen zu können. Ein Parsingalgorithmus muss hier extra auf mögliche Zyklen achten. D.h. er muss mitprotokollieren, welche Produktionen schon ausgeführt wurden und die Zwischenrepräsentationen mitspeichern, um die neuen Zwischenergebnisse mit ihnen vergleichen zu können. Der von

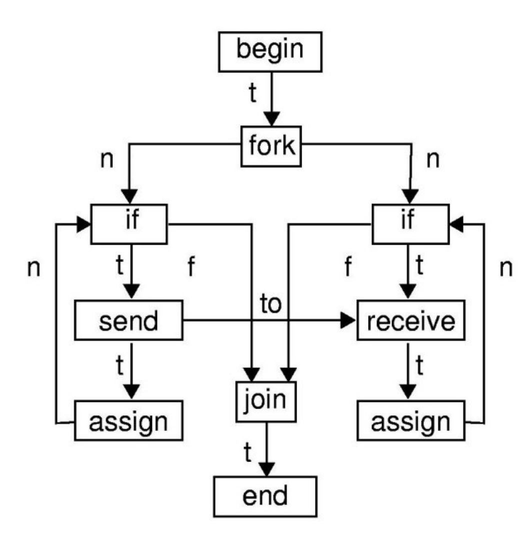


Abbildung 6.2.: Process-Flow-Diagram

Rekers und Schürr entwickelte Algorithmus kann also diese Zykluserkennung einsparen und gewinnt dadurch an Performanz. Der Algorithmus wird in Abschnitt 6.4.2 beschrieben.

6.4 Parsen

Sinn und Zweck einer Grammatik besteht immer darin, einen Formalismus bereitzustellen, mit dem eine Sprache genau bestimmt werden kann. Das ist bei Grammatiken für Programmiersprachen genauso der Fall, wie bei Graphgrammatiken für ER-Diagramme, Abstrakte Syntaxgraphen etc.

Grammatiken für Zeichenketten werden vom Compiler einer Programmiersprache dazu benutzt, Quellcode auf syntaktische Korrektheit zu prüfen. Bei Graphgrammatiken verhält es sich genauso. Hier wird geprüft, ob der Graph eine gewisse, ihm auferlegte Syntax einhält. Wenn nun ein Graph die geforderte Syntax einhält, dann ist er ein Wort der Sprache der Grammatik.

Es gilt also, das Wortproblem zu entscheiden. Ist ein Graph ein Wort der Sprache einer Grammatik, dann muss er durch Anwenden der Produktionen dieser Grammatik aus dem Axiom erzeugt werden können. Die Reihenfolge, in der die Produktionen angewendet werden, ist unerheblich. Es gibt prinzipiell zwei Wege, das Wortproblem zu lösen: Um aus dem Axiom den Graphen herzuleiten, müssen alle Wörter der Sprache aufgezählt werden, d.h. alle möglichen Kombinationen von Produktionen werden „ausprobiert“. Die Wörter müssen dann noch mit dem zu überprüfenden Graphen verglichen werden. Aus dem zu überprüfenden Graphen das Axiom herzuleiten, ist jedoch der einfachere und effizientere Weg. Am Graphen lassen sich recht leicht Positionen finden, an denen Produktionen rückwärts – also von rechts nach links – ausgeführt werden können.

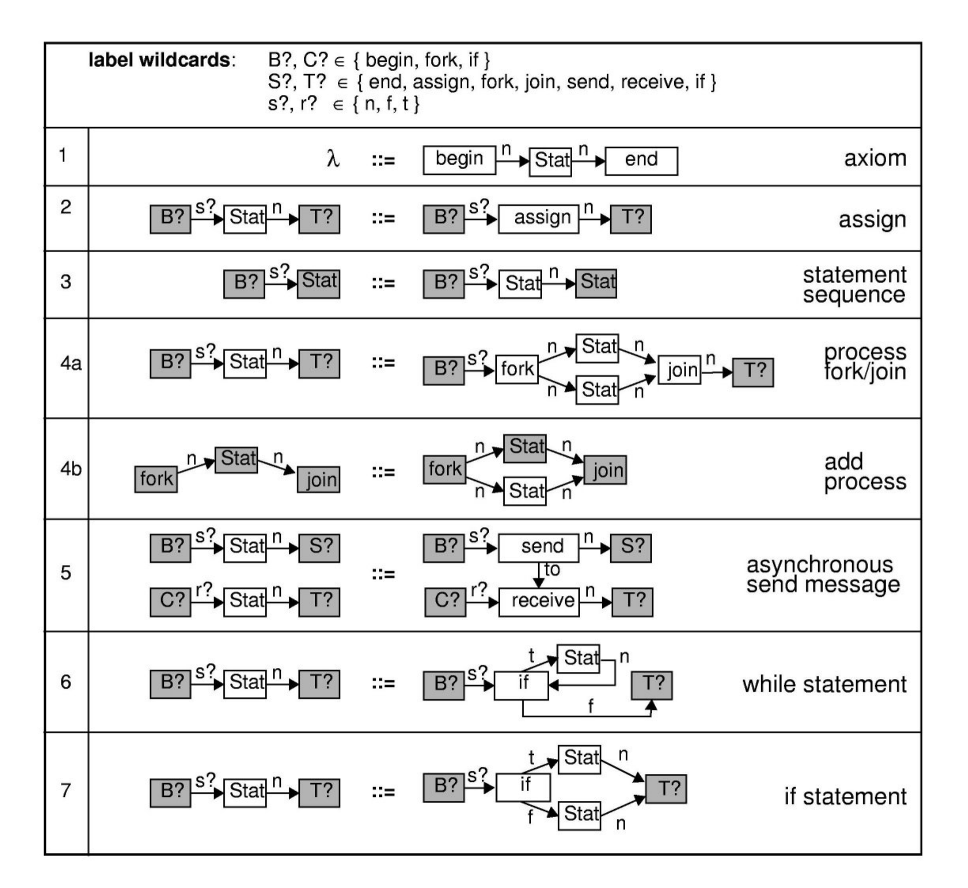


Abbildung 6.3.: Graphgrammatik für Process-Flow-Diagrams

6.4.1 Beispiel

In Abbildung 6.2 ist ein Process-Flow-Diagram (PFD) dargestellt. PFDs besitzen lineare Abläufe von Aktionen, die durch *if*- und *while*-Schleifen gesteuert werden können. Es können darüberhinaus noch parallele Abläufe gebildet werden. Alle diese möglichen Konstrukte sind im Graph in Abbildung 6.2 vorhanden. Die Graphgrammatik für PFDs ist in Abbildung 6.3 dargestellt. Sie kommt mit lediglich sieben Produktionen aus, die alle nötigen Elemente erzeugen können. Die in der Abbildung grau unterlegten Knoten sind dabei Kontextknoten, die vor und nach Anwenden der Produktion unverändert vorhanden sind. Neu hinzugekommene Knoten sind weiß dargestellt.

Um nun aus dem Graphen aus Abbildung 6.2 das Axiom herzuleiten, muss nach Knotenkonstellationen gesucht werden, die der rechten Seite einer Produktion entsprechen. Das ist für Produktion 5 der Fall. Danach kann in beiden parallelen *while*-Schleifen jeweils einmal die Produktion 3 und einmal die Produktion 2 rückwärts angewendet werden. Anschließend lassen sich die *while*-Schleifen mit Produktion 6 entfernen. Das verbleibende *fork-join*-Konstrukt

verschwindet durch umgekehrtes Anwenden der Produktion 4a. Nun besteht der Graph nur noch aus dem Startgraphen, der mit Produktion 1, dem Axiom, aus dem leeren Graph erzeugt wird. Der gegebene Graph läßt sich also auf das Axiom zurückführen. Somit ist das Sprachproblem positiv entschieden und der Graph ist ein Wort der Sprache der Grammatik und stellt ein PFD dar.

6.4.2 Parsingalgorithmus nach Rekers und Schürr

Das obige Beispiel ist natürlich sehr einfach gehalten. Es gibt keine Möglichkeit, zwei verschiedene Produktionen auszuführen, die die gleichen Knoten beträfen und sich somit ausschließen würden. In den beiden *while*-Schleifen ist es zwar möglich, zuerst eine Schleife komplett zurückzuführen und anschließend die andere, oder erst die *assign*-Statements in beiden Schleifen zu entfernen und danach die Schleifen, aber diese Produktionen schließen sich nicht gegenseitig aus.

Wenn sich zwei Produktionen gegenseitig ausschließen, ist nicht klar, welche von beiden nun die „richtige“ ist, sprich welche umgekehrt ausgeführt werden muss, um zum Axiom zu gelangen. Es müssen daher beide Möglichkeiten berechnet werden. Eine Lösung, den richtigen Weg zu finden, wäre es, eine Tiefensuche durchzuführen. Bei großen Graphen gibt es unter Umständen sehr viele Stellen, an denen eine Produktion umgekehrt angewendet werden kann. Die Tiefensuche berechnet daher eine Zwischendarstellung des Graphen mehrfach. Im ersten Suchvorgang wird beispielsweise erst eine Produktion A, anschließend eine Produktion B und danach eine Produktion C ausgeführt, die alle unabhängig voneinander verwendet werden können. In einem weiteren Suchvorgang würde dann erst Produktion B, dann A und danach C ausgeführt. Das Zwischenergebnis wäre das gleiche wie in der ersten Suche. Der Algorithmus würde die für dieses Zwischenergebnis schon vorher – erfolglos – durchgeführte Tiefensuche noch einmal berechnen.

Der zweiphasige Algorithmus, den Rekers und Schürr vorschlagen, verfolgt daher den Ansatz der Breitensuche.

1. Von einem bestehenden Graph werden zunächst in einer *Bottom-Up*-Phase alle Produktionen bestimmt, die umgekehrt ausgeführt werden können. Durch umgekehrtes Anwenden einer Produktion entsteht eine sogenannte Produktionsinstanz PI. Die im ersten Durchgang entstandenen PIs werden nun ihrerseits auch wieder überprüft. Das ganze wiederholt sich solange, bis die einzelnen PIs so weit reduziert worden sind, dass keine Produktionen mehr rückwärts angewendet werden können. Bei der Suche auftretende doppelte PIs werden erkannt und nur einmal abgespeichert. Am Ende der *Bottom-Up*-Phase ist dann eine Sammlung aller Produktionsinstanzen entstanden, die mit PPI bezeichnet wird. Zwischen den einzelnen Produktionsinstanzen bestehen noch Abhängigkeiten. So kann es sein, dass PI_i nur dann erzeugt werden kann, wenn zuvor PI_j erzeugt wurde, oder dass sich PI_k und PI_l gegenseitig ausschließen. Diese Abhängigkeiten werden während dieser ersten Phase des Algorithmus berechnet und mit den Produktionsinstanzen zusammen abgespeichert.

2. In der nun folgenden *Top-Down*-Phase werden die in der *Bottom-Up*-Phase gefundenen Produktionsinstanzen PI_i so zusammengesetzt, dass sie aus dem Graph das Axiom herleiten. In einer Tiefensuche wird dazu eine Teilmenge von PPI gebildet, die genau die

Instanzen enthält, mit denen aus dem Axiom der Graph gebildet werden kann. Dabei werden die verschiedenen Abhängigkeiten zwischen den einzelnen Produktionsinstanzen beachtet, um unnötige bzw. unsinnige Kombinationen verschiedener Instanzen zu eliminieren. Diese würden ohnehin nicht zum Ziel führen.

Läßt sich nun eine solche Teilmenge finden, so ist das Sprachproblem erfolgreich gelöst und der Algorithmus wird erfolgreich abgeschlossen. Je nach Implementierung kann der Algorithmus einen Booleschen Wert zurückgeben, der angibt, ob der überprüfte Graph zur Sprache gehört (*true*) oder nicht (*false*). Eine andere Möglichkeit ist, die gefundene Teilmenge von *PPI* zurückzugeben, wenn der überprüfte Graph enthalten ist; ist er nicht enthalten, so wird die leere Menge zurückgegeben. Der Berechnungsaufwand für beide Rückgabevarianten ist identisch, da die Teilmenge ohnehin berechnet werden muss. Die zweite Variante bietet sich aber an, wenn die ausgeführten Produktionen im weiteren Verlauf des Programmes noch benötigt werden.

6.5 Fazit und Ausblick

Graphgrammtiken sind ein gutes Mittel, Graphen auf ihre syntaktische Korrekheit zu prüfen. Allerdings sind Algorithmen, die das Prüfen letztendlich durchführen, nicht sehr einfach zu realisieren. Algorithmen, die Grammatiken auf Zeichenketten überprüfen, lassen sich recht effizient programmieren. Algorithmen für Graphgrammatiken müssen jedoch noch das Einbettungsproblem beachten. Hierdurch entstehen zum einen beträchtliche Performanzeinbußen, zum anderen steigen die Komplexität des Algorithmus und sein Implementierungsaufwand. Es muss also abgewägt werden, ob dies alles zur Prüfung der Software-Strukturen im Editor-Framework genutzt werden soll, oder ob die vom Anwender durchgeführten Änderungen am Diagramm direkt in Quellcode übersetzt werden, um sie dann vom Compiler der verwendeten Programmiersprache prüfen zu lassen. Dabei muss dann allerdings die Ausgabe des Compilers noch entsprechend analysiert werden.

KAPITEL 7

Der deklarative Ansatz

Sven Wenzel

7.1 Einleitung

Visuelle Sprachen werden in der Informatik zunehmend verwendet. Zum Beispiel eignen sich *Petrinetze* (Ghezzi u. a., 1999, Kapitel 5), um Programmabläufe zu simulieren, während sich *UML-Klassendiagramme* (Hitz und Kappel, 2002, Kapitel 2.1) hervorragend anbieten, um Softwarestrukturen zu modellieren. Die Diagramme werden hier zum Austausch von Informationen genutzt. Zwei Softwareentwickler können so – unabhängig ihrer Herkunft – sicherstellen, dass sie bei einer Unterhaltung über die gleichen Dinge sprechen – vorausgesetzt beide beherrschen diese visuelle Sprache. Darüber hinaus ist es ebenfalls möglich, nicht-visuelle Sachverhalte in Grafiken zu konvertieren, oder aber gegebene Grafiken weiter zu verarbeiten. So kann beispielsweise aus einem UML-Klassendiagramm heraus Quellcode erzeugt werden.

Damit dies funktioniert, benötigen visuelle Sprachen genauso wie natürliche oder Programmiersprachen eine klare Definition ihrer Syntax und ihrer Semantik. Die Spezifikation der Beziehung zwischen einer Grafik und ihrer Bedeutung ist daher das Kernproblem der visuellen Sprachen (Helm und Marriott, 1991) und wird in der Informatik durch verschiedene Ansätze, wie zum Beispiel *Graphgrammatiken* (Schürr und Westfechel, 1992) oder *Contraint Multiset Grammars* (Helm u. a., 1991) angegangen. Ein weiterer Ansatz ist der deklarative. Seine Funktionsweise unterscheidet sich jedoch deutlich von grammatikalischen Ansätzen, wie in Kapitel 7.5 gezeigt wird.

Zunächst soll in Kapitel 7.2 der deklarative Ansatz an einem kleinen Beispiel vorgestellt werden. Anhand dieses Beispieles werden dann im darauffolgenden Kapitel die Eigenschaften beschrieben. Kapitel 7.4 wird sich mit den Funktionsweisen des deklarativen Ansatzes bei der Erzeugung und der Erkennung von Grafiken befassen.

7.2 Einleitendes Beispiel

Das Diagrammm in Abbildung 7.1 zeigt ein *Anwendungsfalldiagramm* der UML (Hitz und Kappel, 2002, Kapitel 2.3). Wir sehen einen Actor *Sven*, der ein Seminar vorbereitet. Dieser Anwendungsfall beinhaltet wiederum die Generierung von Folien durch das technische System *PowerPoint*.

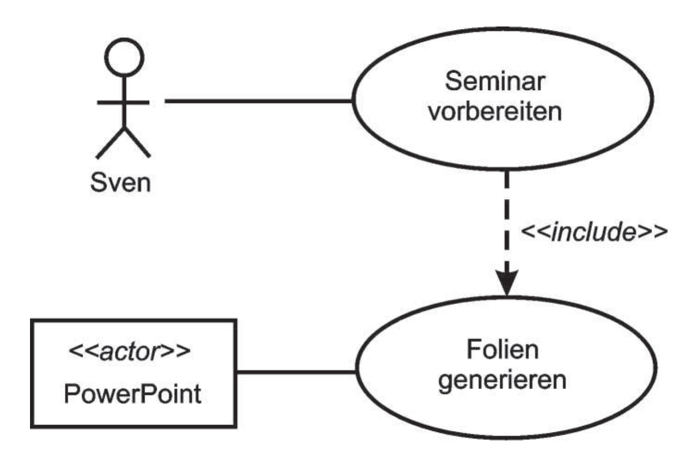


Abbildung 7.1.: UML Anwendungsfalldiagramm

Für einen in der UML geübten Softwareentwickler ist die Aussage dieser Grafik schnell zu überblicken. Sollte nun jemand anderes oder gar ein Computer dieses Diagramm interpretieren, so ist es notwendig, dass die Symbole der Grafik sinngemäß erkannt und zugeordnet werden. Hierzu müssen die Syntax des Diagrammtyps sowie dessen Semantik richtig erkannt werden. Ebenso benötigt man die Kenntnis über Syntax und Semantik eines Diagrammtyps, um ein entsprechendes Diagramm zu erzeugen.

Ohne das Wissen über Syntax und Semantik würde sich für den Betrachter oder ein Programm folgendes Bild lediglich auf Basis von Koordinaten ergeben (hier für den Actor Sven):

- Kreis an Position (1,4) mit Durchmesser 1
- Linie von (1,4) nach (4,4)
- Linie von (3,3) nach (3,5)
- Linie von (3,5) nach (4,4)
- Linie von (4,4) nach (5,5)

Entscheidend ist, dass die Interpretation dieser Linien und Kreise als das zusammenhängende Symbol *Strichmännchen* bereits ein Verständnis der Syntax erfordert und somit nicht trivial ist. Ferner ist die Zuordnung dieses *Strichmännchens* zu einem Akteur der realen Welt, der hier eine Aktivität ausführen soll, ohne den Begriff der Semantik nicht durchführbar.

Aus diesem Grund werden visuelle Sprachen eingesetzt, die als Regelwerk für das Aussehen von Grafiken verwendet werden und die Beziehung zwischen Grafiken und ihren Aussagen spezifizieren. Dabei möchte man ein möglichst einfaches Modell verwenden, welches jedoch auch die Verwendung komplexer und hierarchisch aufgebauter Grafiken erlauben soll. Darüber hinaus soll die visuelle Sprache die Semantik einer Grafik als möglichst formale Beschreibung klar verdeutlichen. Zudem soll sie sich – unabhängig von ihrer Implementierung – dazu eignen, Grafiken zu erzeugen oder gegebene Grafiken zu erkennen.

Die Anforderungen an eine visuelle Sprache lassen sich demnach wie folgt festhalten:

- Unterstützung komplexer Grafiken
- Wiederverwendung von Grafiken als Teile neuer Grafiken (hierarchische Komposition)
- einfache Formulierung topologischer, geometrischer und semantischer Beziehungen zwischen einzelnen Teilgrafiken
- unterstützendes Werkzeug zur Erzeugung und Erkennung von Grafiken
- implementierungsunabhängig

7.3 Eigenschaften

Eine Möglichkeit, Syntax und Semantik für einen Diagrammtyp festzulegen, bietet der deklarative Ansatz. Hierzu gibt verschiedene Möglichkeiten, die deklarativen Syntax- und Semantikdefinitionen textuell aufzuschreiben. So wäre beispielsweise die Verwendung von *XML* möglich. Das Beispiel aus Abbildung 7.2 orientiert sich in der Notation an (Esser und Janneck, 2001) und zeigt ansatzweise, wie die Syntax- und die Semantikdefinition für den Diagrammtyp aus Abbildung 7.1 aussehen kann. Dabei wird jedoch keine vollständige Deklaration von Anwendungsfalldiagrammen gegeben, sondern sich auf den Teil beschränkt, mit dem das Diagramm aus Abbildung 7.1 beschreibbar ist.

Auf den ersten Blick wird deutlich, dass hier eine Beschreibung – in anderen Worten eine Deklaration – des Diagrammtyps vorliegt. Wir gehen im Beispiel davon aus, dass wir einen Grafiktyp in Form eines Graphen wünschen, was durch das Schlüsselwort `graph type` in Zeile 1 signalisiert wird. Anschließend folgt in den Zeilen 2–7 eine Auflistung verschiedener Knotentypen, die uns zur Verfügung stehen, sowie die Nennung möglicher Kantentypen in den Zeilen 8–11. Nach diesem beschreibenden Teil werden einige Bedingungen genannt, die das Diagramm zu erfüllen hat (Zeilen 13–24). So wird zum Beispiel in den Zeilen 21–24 gefordert, dass eine Include-Beziehung nur zwischen zwei Anwendungsfällen bestehen kann.

Im nachfolgenden soll diese Deklaration unter der Herausstellung ihrer wesentlichen Eigenschaften genauer untersucht werden.

Wie sich leicht erkennen lässt, besteht die Deklaration aus zwei grundlegenden Bestandteilen. Der erste Teil beschäftigt sich mit der Beschreibung der einzelnen grafischen Elemente. Dabei ist man jedoch nicht auf ein solch hohes Abstraktionsniveau unseres Beispiels festgelegt. Die Beschreibung der grafischen Elemente ist schachtelbar und somit hierarchisch strukturiert. So verweist der Knotentyp `ActorUser` auf eine Subgrafik `Stickman`, der die Parameter Höhe 30 und Breite 20 mitgegeben werden.

Die Subgrafik `Stickman`, wie in Abbildung 7.3 gezeigt, wird wiederum aus den Subgrafiken Kreis und Linie zusammengesetzt. Auch hier erkennt man wieder eine Angabe von von Bedingungen, die erfüllt werden müssen. In diesem Fall wird geprüft, dass ein Strichmännchen immer im positiven Koordinatenbereich gezeichnet wird und seine Höhe größer als seine Breite ist.

Der zweite grundlegende Bestandteil einer Deklaration ist somit ein Block von Bedingungen, den sogenannten *constraints*. Diese stellen bestimmte Rahmenbedingungen sicher, die für eine Grafik oder Teilgrafik immer erfüllt sein müssen. Auf diesem Weg kann die Syntax

```
graph type UseCaseDiagram {
    vertex type ActorUser(String name)
        graphics(Shape = "Stickman", ExtendX = 30, ExtendY = 20);
    vertex type ActorOther(String name)
        graphics(Shape = "Rectangle", ExtendX = 30, ExtendY =50);
    vertex type UseCase(String title)
        graphics(Shape = "Oval", ExtendX = 30, ExtendY = 70);
    edge type Relation()
        graphics(Style = "Solid");
    edge type Include()
        graphics(Style = "Dashed", Head = "Triangle", Label = "<<include>>");

    // Relationen nur von Actor zu UseCase
    predicate Relation1
    forall r in Relation :
        (start(r) in ActorUser) & (end(r) in UseCase)
    end;

    // Relationen nur von Actor zu UseCase
    predicate Relation2
    forall r in Relation :
        (start(r) in ActorOther) & (end(r) in UseCase)
    end;

    // Include-Beziehungen nur zwischen UseCases
    predicate Include
    forall i in Include :
        (start(i) in UseCase) & (end(i) in UseCase)
    end;
}
```

Abbildung 7.2.: Deklaration des Anwendungsfalldiagramms

```
graphic Stickman(X , Y , H , W) {
    circle(X+(W-X)/2 , Y+(W-X)/4 , (W-X)/4) &
    line(X+(W-X)/2 , Y+(W-X)/2 , X+(W-X)/2 , Y+W) &
    line(X , Y+(H-Y)/2 , X+W , Y+(H-Y)/2) &
    line(X , Y+H , X+(W-X)/2 , Y+W) &
    line(X+W , Y+H , X+(W-X)/2 , Y+W)
    <--
    (X >= 0) & (Y >= 0) & (H > W);
}
```

Abbildung 7.3.: Deklaration des Strichmännchens

einer Grafik formal beschrieben werden. Die Bedingung `Include` aus Abbildung 7.2 stellt also sicher, dass Include-Beziehungen nur zwischen zwei Anwendungsfällen bestehen und nicht etwa zwischen zwei Akteuren.

Es ist offensichtlich, dass der deklarative Ansatz bereits die ersten Anforderungen an eine visuelle Sprache vollständig erfüllt. Hierarchien werden unterstützt, wie das Beispiel des Strichmännchens mit seiner Verschachtelung in Kreise und Linien verdeutlicht hat. Zudem wird deutlich, dass das Strichmännchen mehrfach wiederverwendet werden kann, um komplexere Grafiken zu erzeugen, ohne dass es hierzu erneut definiert werden muss. So wird zum Beispiel auch ein Rechteck einmal definiert und in unzähligen Grafiken verwendet, ohne dass man sich Gedanken darüber machen muss, dass das Rechteck aus vier Linien besteht oder dass die jeweils gegenüberliegenden Linien parallel bzw. die benachbarten Linien senkrecht zueinander sind. Darüber hinaus wird der deklarative Ansatz der Anforderung gerecht, dass topologische, geometrische und semantische Beziehungen zwischen Teilgrafiken formuliert werden können. So wird in unserem Beispiel die Anordnung von Akteuren zu Anwendungsfällen durch die Bedingungen `Relation1` und `Relation2` deutlich gemacht. Der eigentliche Nutzen dieser Bedingungen wird im nachfolgenden Kapitel bei der Erkennung von Grafiken erläutert.

Letztlich hat die Deklaration über all die Anforderungen hinaus den Vorteil, dass je nach ihrer Formulierung schon auf den ersten Blick deutlich werden kann, wie der hier beschriebene Diagrammtyp auszusehen hat bzw. welche Form eine Grafik der visuellen Sprache *Anwendungsfalldiagramm* haben könnte. Gegebenenfalls kann sich der Betrachter der Sprache hier bereits eine bildliche Vorstellung machen.

7.4 Erzeugung und Erkennung von Grafiken

Der erste Teil einer Deklaration beschreibt, *was* später zu sehen sein wird. Der zweite Teil – also die Bedingungen – besagen dabei, *wie* die Grafiken aus der Spezifikation zu erzeugen bzw. zu erkennen sind (Helm und Marriott, 1991) und helfen bei genau diesen Vorgängen, wie im folgenden wieder anhand des Beispiels aus Abbildung 7.1 gezeigt werden soll.

7.4.1 Erzeugung

Es soll der Fall angenommen werden, dass wir die Beziehung zwischen dem Akteur *Sven* und dem Anwendungsfall *Seminar vorbereiten* zeichnen wollen. Wir betrachten die Teilgrafiken *Akteur* und *UseCase* als gegeben und bereits in unsere Zielgrafik eingezeichnet. Wir legen das Augenmerk auf die grafische Repräsentation der *Relation*. Hierzu verwenden wir die Deklaration aus Abbildung 7.4, die der des Strichmännchens ähnlich ist. Der Relation wird als Parameter mitgegeben, dass sie von einem bereits gezeichneten Startobjekt zu einem ebenfalls schon gezeichneten Zielobjekt verlaufen soll. In unserem Fall sind das der Akteur *Sven* und der Anwendungsfall *Seminar vorbereiten*. Diese sind bereits gezeichnet worden, so dass ihre Positionen bekannt sind. Nun werden die deklarierten Bedingungen der Relation geprüft. Entweder muss die Relation also von einem Akteur zu einem Anwendungsfall führen oder umgekehrt. Dazu wird im Beispiel die Funktion `isType()` verwendet, die prüft, ob das im ersten Parameter übergebene Objekt von dem im zweiten Parameter übergebenen Typ ist. Die-

```
graphic Relation(vertex start, vertex end) {
    (isType(start, Actor) & isType(end, UseCase)) |
    (isType(start, UseCase) & isType(end, Actor))
    -->
    line(start, end, "solid");
}
```

Abbildung 7.4.: Deklaration der Relation

se Funktion ist jedoch nur beispielhaft. Funktionen sind frei definierbar und stellen wiederum zusammengefasste Bedingungen dar, die zur mehrfachen Verwendung nicht immer neu definiert werden müssen. Ihre Struktur ist also vergleichbar mit der der Grafiken, die sich aus Teilgrafiken zusammensetzen. Sind die Bedingungen alle erfüllt, so wird das Zeichnen der Subgrafiken angestoßen. Innerhalb dieser wird dann analog vorgegangen. So wird hier das Zeichnen der Linie aufgerufen. Als Parameter werden in diesem Fall der Start- und der Endpunkt der Linie sowie ein Linientyp mitgegeben. Beim Zeichnen der Linie wird zum Beispiel geprüft, ob der Start- und der Endpunkt tatsächlich zwei verschiedene Punkte sind und das Zeichnen der Linie möglich ist.

Das Zeichnen einer Grafik geschieht also im Allgemeinen wie folgt:

1. Prüfen, ob alle Bedingungen erfüllt sind

2. Im Erfolgsfall: Rekursives Zeichnen der Subgrafiken

Bei diesem Vorgehen erfolgt das Zeichnen einzelner Teilgrafiken, wie zum Beispiel eines Akteurs, relativ freizügig. Das Strichmännchen könnte zunächst an beliebiger Position gezeichnet werden. Erst durch die Prüfung der einzelnen Bedingungen wird das Zeichnen verifiziert und anschließend ausgeführt. So könnte zum Beispiel ein Akteur *innerhalb* einer Anwendungsfallblase gezeichnet werden. Erst die bisher nicht vorgestellte Bedingung, dass die Zeichenposition nicht bereits durch ein anderes Objekt belegt sein darf, verhindert, dass das Strichmännchen in das bereits bestehende Oval gezeichnet wird. Es kann also auf der einen Seite relativ frei gezeichnet werden, während auf der anderen Seite die Bedingungen noch während des Zeichnens geprüft werden und die Ausführung unter Umständen verhindert wird.

7.4.2 Erkennung

Wie beim Erkennen einer vorliegenden Grafik vorgegangen wird, soll an dem Beispiel aus Abbildung 7.5 erläutert werden. Das Vorgehen wird hierbei unabhängig von einer konkreten Implementierung betrachtet, da die Suche nach einer wirklich effizienten Umsetzung immer noch Ziel der Forschung ist (Helm und Marriott, 1991).

Ein mögliches Vorgehen wäre, die gegebenen Beschreibungen der Deklaration herzunehmen, schablonenähnlich auf die gegebene Grafik zu legen und mit dieser zu vergleichen. Aufgrund des exakt beschriebenen Aussehens innerhalb der Deklaration kann hier ein guter Vergleich zwischen der deklarierten Vorgabe und der gegebenen Grafik gezogen und diese

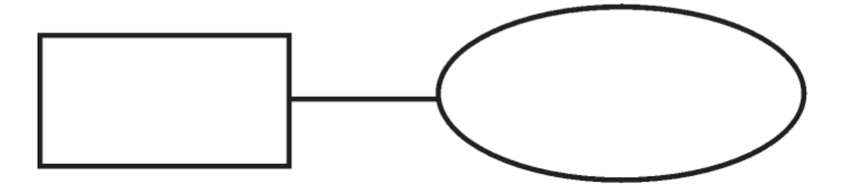


Abbildung 7.5.: Beispielgrafik zu Erkennung

wiederum relativ einfach erkannt werden. So lässt sich erkennen, dass hier ein Rechteck und ein Oval vorliegen, welche durch eine Linie miteinander verbunden sind. Der Vergleich mit den Deklarationen ergibt, dass es sich bei dem Rechteck eindeutig um einen Akteur und bei dem Oval um einen Anwendungsfall handelt.

Das Erkennen der Linie an sich ist dann keine Besonderheit mehr. Es soll vereinfachend angenommen werden, dass sämtliche Linien in einem Anwendungsfalldiagramm – also *Relations*-, *Include*- und *Extend-Beziehungen* gleich aussehen. Dann stellt sich die Frage, welche Art von Linie in dieser Grafik vorliegt und welche Bedeutung sie für die Aussage der Grafik hat. Auch hier ist eine Rückführung auf die Deklaration möglich. Während die Formen aus den Beschreibungen der Deklaration erkannt wurden, erfolgt die Erkennung der Bedeutung aus den Bedingungsteilen der der Deklaration. Nachdem die beiden Objekte, die die Linie verbindet, als Akteur und Anwendungsfall identifiziert worden sind, ist nur noch die Bedingung der Relation (Abbildung 7.4) erfüllt. Die Bedingung einer Include-Beziehung, dass Start- und Zielpunkt der Linie jeweils Anwendungsfälle sein müssen, ist zum Beispiel nicht erfüllt. Daraus kann abgeleitet werden, dass es sich bei dieser Kante um eine Relation handeln muss. Es liegt in Abbildung 7.5 also ein Akteur vor, der zu einem Anwendungsfall in Beziehung steht.

7.5 Abgrenzung gegen Graphgrammatiken

Eine weitere Möglichkeit, visuelle Sprachen zu definieren, bieten Graphgrammatiken (Schürr und Westfechel, 1992). Aus diesem Grund sollen die Funktionsweisen der beiden Definitionsmöglichkeiten verglichen werden, um den deklarativen Ansatzes gegen die Graphgrammatiken abzugrenzen.

Um den Vergleich sinnvoll durchführen zu können, beschränken wir uns auf die Erzeugung und die Erkennung von Graphstrukturen. Graphgrammatiken eignen sich – wie ihr Name schon sagt – besonders gut zur Definition von Graphstrukturen. Dabei stehen verschiedene Kanten- und Knotentypen zur Verfügung, so dass sich auch komplexe Diagramme, wie zum Beispiel ER-Diagramme, mit Hilfe von Graphgrammatiken definieren lassen. Der deklarative Ansatz hingegen behandelt nicht ausschließlich Graphstrukturen, sondern Grafiken, und ist damit an dieser Stelle etwas mächtiger. Während Graphen aus Knoten und Kanten verschiedenster Arten bestehen, kann eine Deklaration auf die Menge aller geometrischer Figuren und beliebige Grafiken zurückgreifen. Dementsprechend ist die Funktionsweise der Graphgrammatik von der Funktionsweise der Deklaration wie folgt zu unterscheiden.

7.5.1 Produktion vs. Deklaration

Graphgrammatiken sind mit Stringgrammatiken vergleichbar und arbeiten mit einer Menge von Produktionen, die auf Graphen angewendet werden. Eine Produktion ist dabei die Zuordnung einer Graphstruktur A auf der linken Seite zu einer anderen größeren Graphstruktur B auf der rechten Seite. Sie ist als eine Art Regel zu verstehen, mit der eine in dem Graphen vorhandene Struktur A durch die Struktur B ersetzt werden darf. Die Graphstrukturen umfassen dabei sowohl Terminal- als auch Nicht-Terminal-Symbole. Bei der Erzeugung eines Graphen wird von einem Startsymbol – meist einem leeren Graphen – ausgegangen. Dieser wird anschließend mit Hilfe der Produktionen zu dem gewünschten Graphen geformt.

Durch die Vorgabe der Produktionen ist für das Erzeugen eines Graphen eine recht genau definierte Vorgehensweise gegeben. Aufgrund dessen ist der neue Graph zu jedem Zeitpunkt seiner Erzeugung syntaktisch korrekt und somit ein Wort der durch die Grammatik beschriebenen Sprache. Damit unterscheidet sich der deklarative Ansatz deutlich von den Graphgrammatiken. Die Deklarationen geben keine Ordnung zur Grapherzeugung vor, sondern lassen eine willkürliche Reihenfolge der Erstellung zu und prüfen erst während des Zeichnens, ob die Bedingungen alle erfüllt sind.

7.5.2 Erkennung

Bei der Erkennung von Graphen unterscheidet sich der deklarative Ansatz ebenfalls von der Vorgehensweise der Graphgrammatiken. Die einzige Ähnlichkeit besteht darin, dass der deklarative Ansatz die gegebenen Deklarationen und die Graphgrammatiken ihre Produktionen verwenden. Die Graphgrammatiken wenden ihre Produktionen nun in umgekehrter Richtung an. Es wird untersucht, ob irgendwo die rechte Seite einer Produktion mit dem Graphen übereinstimmt. Ist dies der Fall, so wird dieser Teilgraph durch die linke Seite der Produktion reduziert. Dieser Vorgang wird solange wiederholt, bis das Startsymbol der Grammatik erreicht ist. Daraus folgt, dass es sich bei dem Graphen um ein Wort der visuellen Sprache handelt.

Der deklarative Ansatz versucht durch Vergleiche Teilgrafiken herauszufiltern, die einer Deklaration entsprechen. Darüber hinaus werden die Bedingungen der Deklaration geprüft. Eine Grafik wird hier als Wort der visuellen Sprache bezeichnet, wenn eine völlige Aufteilung in Teilgrafiken möglich ist und sämtliche Bedingungen erfüllt sind.

Gegenüber der Graphgrammatik, die prüft, ob ein vorliegender Graph ein Wort der von ihr beschriebenen Sprache ist, beschränkt sich der deklarative Ansatz nicht allein auf die Syntax des vorliegenden Graphen. Wie in Kapitel 7.4.2 am Beispiel der Linie zwischen Oval und Rechteck gezeigt worden ist, gibt die Deklaration nicht nur Aufschluss darüber, dass eine Linie vorliegt, sondern auch dass es sich um eine Relation handelt. Damit ist also auch eine semantische Information durch die Erkennung des Graphen bekannt geworden.

7.6 Fazit

Nach der Untersuchung des deklarativen Ansatzes und seiner Arbeitsweise sowie dem Vergleich mit Graphgrammatiken ist deutlich geworden, dass dieser Ansatz durchaus eine sinnvolle Möglichkeit zur Definition visueller Sprachen ist. Die Deklarationen stellen hierbei eine

gut lesbare Repräsentation einer visuellen Sprache dar. Beim Betrachten der Deklaration kann man sich vorstellen, wie die Wörter bzw. Grafiken der Sprache aussehen werden. Die Deklaration beschränkt sich hierbei nicht auf Graphstrukturen, sondern unterstützt komplexe Grafiken. Bedingt durch die Wiederverwendbarkeit der einzelnen Teilgrafiken ist zudem eine hierarchische Komposition von Elementen möglich, was wiederum die Bildung komplexer Grafiken unterstützt. Die Bedingungen ermöglichen, semantische Beziehungen sowie topologische und geometrische Zusammenhänge zwischen den einzelnen Teilgrafiken zu formulieren. Es sei jedoch kritisch angemerkt, dass die semantische Beziehung von Grafiken zu ihrer inhaltlichen Bedeutung nicht immer eindeutig ist. Bei der Grafikerkennung (Kapitel 7.4.2) wird deutlich, dass dies nur dann möglich ist, wenn sich die Beschreibungen und Bedingungen einzelner Teilgrafiken untereinander jeweils gut voneinander abgrenzen lassen. Die Unterscheidung einer Include- von einer Extends-Beziehung erweist sich zum Beispiel als besonders schwierig, da ihre grafischen Repräsentationen mit Ausnahme der Kantenbeschriftungen äquivalent sind. Hier können sich durch eine unzureichende Deklaration recht schnell Fehler einschleichen. Dennoch eignet sich der deklarative Ansatz, um visuelle Sprachen formal zu fassen. Eine konkrete Implementierung des Ansatzes erweist sich jedoch als teilweise schwierig, da die Algorithmen recht komplex sind. Die Umsetzung der deklarationsbasierten Erkennung ist noch immer Arbeitsgebiet der Forschung (Helm und Marriott, 1991), da für einige Deklarationen eine Top-Down- und für andere wiederum eine Bottom-Up-Erkennung effizienter ist, was eine einheitliche Implementierung erschwert. Zudem fließt hier auch das Problem der Bilderkennung mit ein. Die Erzeugung von Grafiken durch deklarative Sprachdefinitionen ist hingegen umzusetzen, indem die Deklarationen mit Hilfe logischer Programmiersprachen realisiert werden (Helm und Marriott, 1991). Das Kernproblem der Erzeugung ist demnach die Abbildung der Deklaration einer Grafik auf die jeweilige Programmiersprache, in der die Anwendung implementiert wird, die die visuelle Sprache verwendet. So erweist es sich für das Erzeugen als sinnvoll, wenn die Deklaration bereits in der Programmiersprache der Anwendung formuliert ist.

KAPITEL 8

Das Eclipse Plugin Modell

Christian Mocek

8.1 Einführung in Eclipse

Das unter der Schirmherrschaft von IBM entworfene Softwaresystem Eclipse wurde unter dem Aspekt einer offenen, erweiterbaren Architektur entwickelt (Eclipse Foundation, 2003). Ziel ist es hierbei, dem Entwickler ein Werkzeug zur Verfügung zu stellen, um Applikationen auf Basis von Plugins zu entwickeln. In der Regel werden dies Entwicklungsumgebungen für z.B. Java, C#, C++ oder auch LaTeX sein, es können jedoch im Prinzip auch Produkte für einen Endanwender auf Eclipse basieren.

Im Folgenden geben wir zur Einführung zunächst einen kleinen Überblick über die Oberfläche und Terminologie von Eclipse, bevor in Kapitel 8.2 eine Beschreibung des Prinzips der Plugin-Architektur von Eclipse folgt.

8.1.1 Die Workbench

Die *Workbench* dient zur Interaktion zwischen dem Endanwender und der Eclipse Plattform. Sie verwaltet die *Ressourcen* und die Steuerelemente der Plugins. Jede Workbench besteht also neben der Verwaltung der Ressourcen noch aus den drei wesentlichen Komponenten: *Perspektiven*, *Editoren* und *Views*.

Perspektiven

Definition 8.1.1: Perspektive
Eine Perspektive definiert die Anordnung der Editoren und Views in der Workbench, sowie die zur Verfügung stehenden Toolbars und Pull-down Menüs.

Das Ziel eine Perspektive ist es also, eine optimale Anordnung von Editoren und Views bereitzustellen, um eine bestimmte Aufgabe optimal bearbeiten zu können. Das heißt natürlich auch, dass jede Workbench aus mehreren Perspektiven bestehen kann. Beispielsweise stellt die Perspektive für die Java-Entwicklungsumgebung andere Views dar, als die Perspektive für das Debuggen von Java Applikationen.

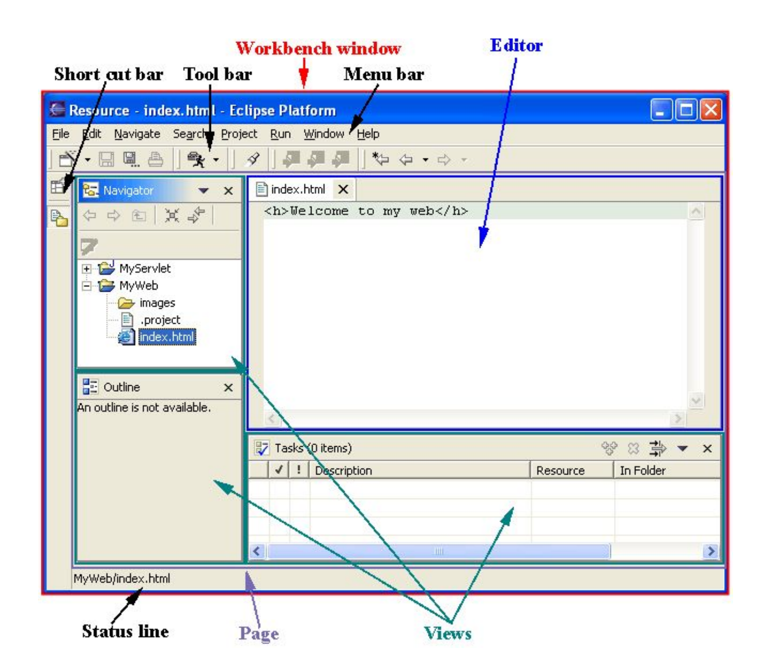


Abbildung 8.1.: Die Workbench von Eclipse.

Editoren

Definition 8.1.2: Editor
Als Editor versteht man den Teil der Workbench, der die Möglichkeit bietet, Dateien eines bestimmten Dateityps zu bearbeiten.

Das heißt also, dass verschiedene Dateitypen einem Editor eindeutig zugeordnet werden können. Dieser Editor ist spezialisiert auf die Bearbeitung der für ihn registrierten Dateitypen. Sollte ein Dateityp keinem Editor zugewiesen sein, wird versucht einen externen Editor, also einen Editor außerhalb der Eclipse Umgebung, zu starten. In der Regel enthalten die meisten Perspektiven genau einen Bereich, in dem die geöffneten Editoren und verschiedene zu den Editoren zugehörige Views dargestellt werden. Falls mehrere Editoren geöffnet sind, können diese in dem zur Verfügung stehenden Bereich via Tabpages ausgewählt werden.

Views

Definition 8.1.3: View
Ein View unterstützt Editoren und bietet alternative Darstellungsweisen der verfügbaren Informationen oder hilft bei der Navigation und Verwaltung von Workbenchinformationen.

Beispielsweise unterstützt der *Navigator-View* die oben beschriebene Verwaltung von Dokumenten in der Workbench, während der *Properties-View* Editierfunktionen von Objekteigen-

schaften bietet.

Ressourcen

Die Struktur der Ressourcen wird im so genannten *Navigator* angezeigt. Von diesem aus kann auf die einzelnen Ressourcen zum Bearbeiten zugegriffen werden. Die Workbench verwaltet drei verschiedene Typen von Ressourcen:

1. Dateien,
2. Verzeichnisse und
3. Projekte.

Dateien und Verzeichnisse entsprechen den Dateien und Verzeichnissen des zugrunde liegenden Dateisystems. Verzeichnisse können andere Verzeichnisse und Dateien enthalten oder auch in Projekten liegen. Projekte selbst sind immer „die Wurzel eines Projektbaums" und können ausschließlich Verzeichnisse und Dateien enthalten, aber keine anderen Projekte. Ähnlich wie Verzeichnisse wird beim Erzeugen eines Projektes ein Pfad auf das zugrunde liegende Verzeichnis in dem Dateisystem spezifiziert.

8.1.2 Die Architektur der Eclipse Plattform

Wie zu Anfang erwähnt, ist die Eclipse-Plattform unter dem Aspekt einer offenen Architektur entwickelt worden (Eclipse Foundation, 2003). Dies wurde dadurch erreicht, dass es im Wesentlichen einen kleinen Plattformkern gibt, der dafür Sorge trägt, Plugins zu starten und zu verwalten. Die oben beschriebene Workbench ist somit nur ein Teil der gesamten Eclipse Plattform. In Abbildung 8.2 ist der schematische Plattformaufbau dargestellt. Jede der dort dargestellten Komponenten ist wiederum als Plugin implementiert. Die Funktionalität der Plattform basiert also auf diesen Plugins. In einer Standard Eclipse-SDK Installation sind Plugins für die Ressourcenverwaltung, der grafischen Benutzeroberfläche, dem Hilfesystem, der Teamarbeit mittels CVS und natürlich dem Plattformkern enthalten. Als zusätzliche Plugins, die nicht zur Eclipse Plattform selbst gehören, sind die *JDT*, die Java-Entwicklungsumgebung und das *PDE*, die Plugin-Entwicklungsumgebung, enthalten. Soll z.B. ein Plugin für die Workbench mit Benutzeroberfläche entwickelt werden, greift man unter anderem auf die schon vorhandenen Komponenten, wie z.B. Views und Editoren der Workbench zu. Zusätzlich bietet Eclipse mit dem *Standard Widget Toolkit* und *JFace* eine Alternative zu der von Sun gelieferten Swing/AWT Lösung zur Entwicklung von grafischen Benutzeroberflächen. Vorteil von SWT/JFace liegt in der Geschwindigkeit und dem nativen Look-and-Feel, welches bei Suns Lösung nicht gegeben ist (Daum, 2003).

Jede Eclipse-Installation besitzt ein Verzeichnis Namens `Plugins`, in welchem die Plugins installiert werden. Jedes Plugin besitzt ein eigenes Verzeichnis, in dem die benötigten Ressourcen wie z.B. Icons, Javaklassen und die das Plugin beschreibende *Manifestdatei* `plugin.xml` liegen.

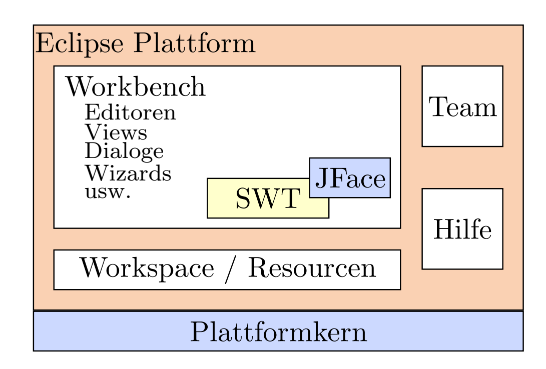


Abbildung 8.2.: Die Komponenten der Eclipse Plattform (Daum, 2003).

Definition 8.1.4: Plugin-Manifest
Als Plugin Manifest bezeichnet man eine Datei, die Informationen über den strukturellen Aufbau des Plugins der Plattform zur Verfügung stellt.

Beim Start von Eclipse werden die Informationen aus den in dem Plugins Verzeichnis liegenden Manifestdateien ausgelesen und in einem Teil des Plattformkerns, der *Plugin Registry*, registriert. Mittels dieses Repositoriums wird bei Bedarf eine Instanz des Plugins erzeugt. Dadurch, dass nur Instanzen der Plugins erzeugt werden, die auch tatsächlich gebraucht werden, wird eine gute Ladezeit der Plattform und ein verbessertes Ressourcenmanagement gewährleistet. Zusätzlich hat der Entwickler über die Plugin Registry API die Möglichkeit, Informationen über die installierten Plugins zu erhalten (Bolour, 2003). Ein wesentlicher (schwacher) Punkt in der Entwicklung von Plugins für Eclipse liegt jedoch darin, dass Plugins nicht während des laufenden Betriebs installiert werden können. Dies liegt an der oben genannten Tatsache, dass die Manifest Dateien nur beim Start von Eclipse gelesen werden. Somit ist der Entwickler gezwungen, eine zweite Workbench zu starten, in der das Plugin getestet werden kann. Allerdings bietet das PDE auch hier gute Möglichkeiten, so dass trotz dieser Tatsache eine effiziente Plugin-Entwicklung gewährleistet werden kann.

Allerdings stellt sich nun die Frage, inwiefern Plugins miteinander kommunizieren und aufeinander aufbauen können. Wegen der offenen Architektur von Eclipse muss es solch eine Möglichkeit geben, sonst muss jeder Plugin-Entwickler „from the scratch“, also von Grund auf, immer alles neu entwickeln. Hierzu stellen wir im nächsten Kapitel das Kernkonzept der Plugin Architektur vor, die so genannten *Extension Points*, und gehen auf die möglichen Relationen zwischen Plugins und deren Aufbau, Beschreibung und Kommunikation untereinander näher ein.

8.2 Das Plugin Modell im Detail

Der zentrale Punkt des Plugin Konzeptes ist die Möglichkeit, anderen Plugins mitteilen zu können,

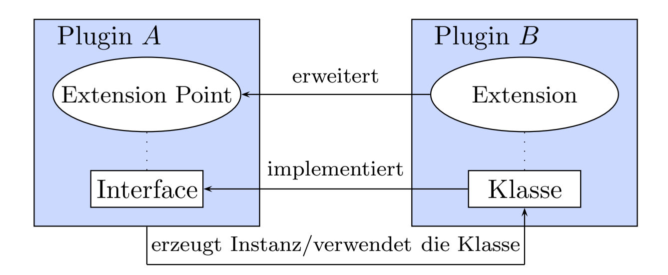


Abbildung 8.3.: Extension-Zusammenhang zwischen zwei Plugins (Eclipse Foundation, 2003).

- um welche Funktionen sie die Plattform erweitern und
- um welche Funktionen das Plugin von anderen Plugins erweitert werden kann.

Um diese Informationen mitteilen zu können, werden die oben genannten Extension Points verwendet.

Definition 8.2.1: Erweiterungspunkt
Ein Erweiterungspunkt definiert für ein Plugin, um welche Funktionalität das Plugin von anderen Plugins erweitert werden kann.

Das hat natürlich zur Konsequenz, dass jedes Plugin mindestens auf einen oder mehrere Erweiterungspunkte zugreift und über diese neue Funktionalität in die Plattform mit einbringen (Eclipse Foundation, 2003). Optional kann ein Plugin ebenfalls eigene Erweiterungspunkte definieren. Somit entsteht ein Netz von Abhängigkeiten der Plugins untereinander, sodass ein Plugin *A* mit einem anderen Plugin *B* in einer der folgenden Relationen stehen kann (Bolour, 2003):

1. *Abhängigkeit:* Das bedeutet, dass *A* das *vorausgesetzte* Plugin ist und *B* das von *A abhängige*. *A* liefert also die Funktionalität, die *B* benötigt, um korrekt arbeiten zu können.
2. *Extension:* Die Rollen in der Relation sind so verteilt, dass *A* das *Basisplugin* ist und *B* das *erweiternde Plugin*. *B* erweitert die Funktionalität von Plugin *A*.

In Abbildung 8.3 ist dieser Zusammenhang verdeutlicht. Plugin *A* deklariert einen Erweiterungspunkt und ein Interface, welches mit dem Erweiterungspunkt verknüpft wird. Da Plugin *B* den Erweiterungspunkt verwendet, implementiert *B* das Interface von *A* in einer Klasse. Diese Klasse erweitert also den Extension Point aus *A* und *A* ruft dann die Methoden des Interface in der Klasse aus *B* auf.

8.3 Das Plugin-Manifest

Einer der wesentlichsten Bestandteile eines Plugins ist die schon erwähnte `plugin.xml`-Datei im „Hauptverzeichnis“ des Plugins. Somit startet die Entwicklung eines Plugins in der Regel

mit der Erstellung solch einer XML-Datei. In dieser Datei werden neben einer allgemeinen Beschreibung des Plugins auch die Relationen deklariert. Die Minimaleinträge der Manifestdatei bestehen aus:

- einem Namen für das Plugin,
- einer *eindeutigen* ID,
- der Versionsnummer des Plugins.

Bei der ID ist zu beachten, dass diese plattformweit eindeutig sein muss. Um dies zu gewährleisten, ist es sinnvoll, den kompletten Paketnamen zu verwenden. In den nächsten Abschnitten werden wir näher auf die Einträge in der Manifest-Datei eingehen. Der folgende XML-Code zeigt, wie eine minimale Manifest-Datei aussieht. Zusätzlich ist in diesem Code vermerkt, wie man den ersten Teil der möglichen Relationen, die Abhängigkeit, deklariert. Hierzu genügt es, die notwendigen Plugins als import-Element in einem requires-Element mit in das Plugin-Manifest mit aufzunehmen. Das runtime-Element definiert die Paketdatei der zu dem Plugin gehörenden Javaklassen.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
    name="PG444␣Beispiel-Plugin"
    id="PG444.pgPlug"
    version="1.0.0"
    provider-name="PG444.org">
    <runtime>
        <library name = "beispiel.jar"/>
    </runtime>
    <requires>
        <import plugin="org.eclipse.ui"/>
    </requires>
</plugin>
```

In den nächsten beiden Abschnitten beschäftigen wir uns mit der Frage, wie die Relation „Extension“ funktioniert, also wie neue Funktionalität der Plattform zur Verfügung gestellt und eigene Erweiterungspunkte definiert werden. Wie in Abbildung 8.3 gezeigt, gibt es in der Relation zwei beteiligte Plugins mit den Rollen des Basisplugins und des erweiternden Plugins.

8.4 Erweiterungspunkte

8.4.1 Deklaration neuer Erweiterungspunkte

Das Basisplugin stellt der Plattform neue Erweiterungspunkte zur Verfügung. Es muss also dafür sorgen, dass andere Plugins von diesen Erweiterungspunkten erfahren, damit diese ihrerseits die Funktionalität des Plugins erweitern können. Die Erweiterungspunkte werden in dem Plugin-Manifest mithilfe des XML-Elements „*extension-point*“ deklariert.

Beispielsweise steht man bei der Entwicklung von Benutzerschnittstellen vor dem Problem, dass Menüs oder Toolbars in die Workbench mit eingebunden werden müssen. Hierzu bietet die Eclipse-Plattform in dem Plugin `org.eclipse.ui` die so genannten *actionSets* an. Der folgende Ausschnitt aus dem Plugin-Manifest beschreibt die Deklaration des actionSets-Erweiterungspunktes.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
    id="org.eclipse.ui"
    name="Eclipse␣UI"
    version="3.0.0"
    provider-name="Eclipse.org"
    class="org.eclipse.ui.internal.UIPlugin">
    <!-- Runtime und Required Elemente -->
    ...
    <!-- Extension Points -->
    <extension-point
        id="actionSets"
        name="Action␣Sets"
        schema="schema/actionSets.exsd"/>
    <!-- hier folgen die restlichen Deklarationen -->
</plugin>
```

Bei der Deklaration von neuen Erweiterungspunkten ist zu beachten, dass die Attribute ID und Name angegeben sein müssen. Die ID muss auch hier wieder innerhalb des Plugins eindeutig sein. Um dies auch global zu gewährleisten, wird der ID die eindeutige Plugin-ID vorangestellt, sodass von anderen Plugins dieser Erweiterungspunkt nur über `org.eclipse.ui.actionSets` angesprochen werden kann (Bolour, 2003). Ein weiteres Attribut der Deklaration ist das Schema. Dies muss nicht explizit angegeben werden, es sei denn die Datei

1. besitzt einen anderen Namen als die id des Erweiterungspunktes, oder aber
2. sie liegt in einem Unterverzeichnis des Plugin-Hauptverzeichnisses.

Im folgenden Abschnitt gehen wir genauer auf diesen wichtigen Teil der Extension-Points ein.

8.4.2 Beschreibung der Erweiterungspunkte

Bei der Erzeugung eines neuen Erweiterungspunktes reicht es nicht aus, diesen in dem Plugin-Manifest zu deklarieren. Das Problem liegt darin, dass erweiternde Plugins wissen müssen, wie die Erweiterungen aussehen müssen, also welche Informationen benötigt werden, um die Erweiterung zu instanziieren. Das Ziel eines Erweiterungsschemas ist es also, einen Erweiterungspunkt zu beschreiben. Ein weiterer Vorteil des Schematas ist, dass aus diesem eine Referenzdokumentation des Erweiterungspunktes automatisch generiert werden kann (Daum, 2003).

Definition 8.4.1: Erweiterungsschema
Das Erweiterungsschema für einen Erweiterungspunkt definiert, welche Informationen das

erweiternde Plugin wie zur Verfügung stellen muss, damit die erweiternden Funktionen vom Basisplugin angesprochen und verwendet werden können. Jedem Erweiterungspunkt wird eindeutig ein Schema zugeordnet.

Die Beschreibung in einem solchen Schema findet in der Sprache XML-Schema statt. Ein Schema besteht im Wesentlichen aus *Elementen* auf die die Erweiterung Zugriff besitzt. Diese Elemente können zusätzlich mit *Attributen* versehen werden. Ein Attribut kann einer der folgenden drei Typen sein:

- Der Typ *java* enthält den Pfad zu einer Javaklasse.
- Der Typ *resource* enthält den Pfad einer Ressource des Eclipse-Workspace.
- Der Typ *string* enthält einen Datenwert. Hierüber werden ebenfalls Boolesche Attribute realisiert.

In manchen Fällen ist es erforderlich, Werte vorzubelegen oder die Angabe zu erzwingen. Dies geschieht mit den Schlüsselwörtern *value* bzw. *use*.

Die Schemastruktur

Wichtig bei der Definition eines Schemas ist dessen struktureller Aufbau. Darunter versteht man die Anordnung der einzelnen Elemente in einer Baumstruktur. Von der Wurzel dieses Baumes müssen dann alle Elemente erreichbar sein. Somit kann ein Element entweder die Wurzel eines Teilbaums sein oder aber ein Blatt. Für jeden Knoten gibt es verschiedene Arten von Verzweigungsmöglichkeiten. Diese so genannten *Konnektoren* sind im Folgenden aufgeführt:

- *Sequenz*: Alle Kinder des Knotens werden in einer *geordneten* Liste organisiert.
- *All*: Es werden alle Kinder des Knotens in einer *ungeordneten* Liste organisiert.
- *Choice*: In einer Instanz darf nur *genau ein* Knoten der Liste angegeben werden.

Den Aufbau der Baumstruktur zeigen wir nun an einem Beispiel. Wird ein neues Schema erzeugt, so generiert man in der Regel zunächst ein allgemeines Element Namens „extension“, welches allgemeine Eigenschaften des Erweiterungspunktes beschreibt (Daum, 2003). Dies ist dann die Wurzel des Baums und spezifische Elemente werden dann an diesem eingehangen. Zusätzlich kann für jedes Element, jedes Attribut und jeden Konnektor mittels *minOccurs* und *maxOccurs* angegeben werden, in welchen Grenzen sich die Anzahl der Wiederholungen befinden darf. Dies wollen wir nun an einem Beispiel verdeutlichen. Der folgende Ausschnitt ist aus der Schemadatei der schon oben erwähnten Action Sets.

```
<schema targetNamespace="org.eclipse.ui">
    <!-- Wurzel des Baums -->
    <element name="extension">
        <complexType>
            <sequence>
                <element ref="actionSet"
```

```
                    minOccurs="1"
                    maxOccurs="unbounded"/>
            </sequence>
            <attribute name="point"
                      type="string"
                      use="required"></attribute>
            <attribute name="id"...
            <attribute name="name"...
        </complexType>
    </element>

    <!-- Element Action Set -->
    <element name="actionSet">
        <complexType>
            <sequence>
                <element ref="menu"
                         minOccurs="0"
                         maxOccurs="unbounded"/>
                <element ref="action"...
            </sequence>
            <attribute name="id"...
            <attribute name="label"...
        </complexType>
    </element>

    <!-- eine Aktion innerhalb des Action Sets -->
    <element name="action">
        <complexType>
            <choice>
                <element ref="selection"...
                <element ref="enablement"...
            </choice>
            <attribute name="id"...
            <attribute name="label"...
        </complexType>
    </element>
</schema>
```

In dieser gekürzten Version der Schemadatei kann man die Baumstruktur ableiten, wie sie in Abbildung 8.4 dargestellt ist. Zusätzlich erkennt man hier den Sinn der minOccurs und maxOccurs Parameter: ein ActionSets Erweiterungspunkt enthält mindestens ein ActionSet, kann aber auch unbegrenzt viele enthalten. Des Weiteren wird jedes der Elemente und Attribute mit einem Kommentar versehen, aus dem später die Referenzdokumentation erstellt werden kann.

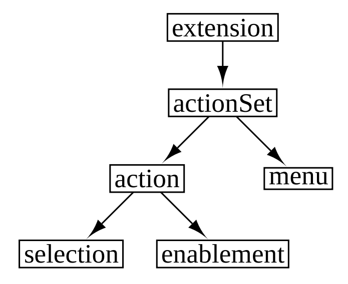


Abbildung 8.4.: Baumstruktur des ActionSet Beispiels.

8.5 Erweiterungen

Um Funktionen zu einem Plugin hinzuzufügen, müssen also die Erweiterungspunkte bekannt und beschrieben sein. Die Aufgabe des Entwicklers ist es nun

- der Plattform bekannt zu geben, dass das entwickelte Plugin neue Funktionen einbindet und
- die entsprechende Funktionalität zu implementieren.

Hierzu müssen die Erweiterungen in dem Plugin-Manifest deklariert werden. Dies geschieht mit dem XML-Element „*extension*“. Mithilfe des folgenden Beispiels wollen wir diesen Vorgang verdeutlichen.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
    name="PG444␣Beispiel-Plugin"
    id="PG444.pgPlug"
    version"1.0.0"
    provider-name="PG444.org">
    ...
    <!-- Action Sets -->
    <extension point="org.eclipse.ui.actionSets">
        <actionSet
            label="Suche␣und␣Hilfe"
            visible="true"
            id="PG444.pgPlug.ActionSet">
            <action
                id="PG444.pgPlug.HelpAction">
                label="&Hilfe"
                class="PG444.pgPlug.HelpContentsAction"
            </action>
            <action
                id="PG444.pgPlug.SearchPage"
                label="&Suche"
                class="PG444.pgPlug.SearchContentAction"
```

```
            </action>
        </actionSet>
    </extension>
    <extension point="org.eclipse.ui.perspectives">
        <perspective
            name="PG444␣Perspektive"
            icon="icons/perspektive.bmp"
            class="PG444.pgPlug.PerspectiveFactory"
            id="PG444.pgPlug.perspective">
        </perspective>
    </extension>
</plugin>
```

Eine Erweiterung wird mit „extension“ eingeleitet und eindeutig mit einem Erweiterungspunkt verknüpft. In dem Beispiel werden zwei Erweiterungen durchgeführt. Zum einen wird eine neue Perspektive zur Workbench hinzugefügt und zum anderen via ActionSets die Menüs in Eclipse um zwei Einträge erweitert. In einer Erweiterung auf einem Erweiterungspunkt können natürlich nur so viele Erweiterungen des Typs stehen, wie das Erweiterungsschema dies über minOccurs und maxOccurs erlaubt. Die Angabe des Attributs „*class*“ sorgt dafür, dass das Basisplugin weiß, welche Klasse das jeweilige Interface für das Element implementiert.

Definition 8.5.1: Callback-Klasse
Ein Plugin besteht in der Regel aus zusätzlichen normalen Javaklassen, die ein bestimmtes Interface eines Erweiterungspunktes implementieren und darüber die Funktionalität zur Verfügung stellen. Diese Klassen heißen Callback-Klassen.

Die Kommunikation des Basisplugins und des erweiternden Plugins findet also über die Callback-Klassen statt. Jedoch hat, wie wir schon gesehen haben, das erweiternde Plugin keinen Einfluss darauf, wann eine Instanz der Callback-Klasse erzeugt wird. Dies wird vom Basisplugin gesteuert.

Allerdings ist es nicht immer für jede Erweiterung notwendig eine Callback-Klasse zu implementieren. Es gibt Elemente, die nur dazu dienen, Information an das Basisplugin weiterzuleiten. Beispielsweise wird für das ActionSet keine Callback-Klasse implementiert, jedoch intern ein Objekt erzeugt. In diesem Fall haben Plugin-Entwickler dann nichts weiter zu tun. Bei der Entwicklung eigener Erweiterungspunkte jedoch muss dann solch eine Klasse implementiert werden. (Bolour, 2003)

8.6 Plugins und Fragmente

In der Regel besteht ein Plugin aus der Plugin Manifest-Datei und den Javaklassen inklusive der notwendigen Ressourcen. Somit bekommt der Anwender ein großes Paket mit dem Plugin (Daum, 2003). Jedoch ist es gerade bei dem Fall der Internationalisierung sinnvoll, das Plugin in Pakete aufzuteilen.

Definition 8.6.1: Fragment
Ein Fragment ist ein vom Plugin unabhängiger Zusatz, der Teile des Plugins enthält.

Fragmente können also unabhängig vom Rest des Plugins entwickelt werden, falls das eigentliche Plugin die Basisfunktionalitäten enthält. Es ist also darauf zu achten, dass das Plugin selbst auch ohne das Fragment läuft. Beispielsweise können bei der Internationalisierung eines Plugins die einzelnen Sprachdateien in Fragmenten untergebracht sein. Besteht ein Plugin aus dem Basisplugin und Fragmenten, so wird es auch als *logisches Plugin* bezeichnet (Eclipse Foundation, 2003). Fragmente sind in eigenen Unterverzeichnissen im Plugin-Verzeichnis untergebracht und besitzen eine eigene Manifest-Datei Namens `fragment.xml`. Das Fragment selbst bildet aber kein eigenes Plugin und implementiert somit nicht eine eigene Plugin-Klasse.

8.6.1 Internationalisierung mit Hilfe von Fragmenten

Zum Schluss wollen wir noch kurz auf die Internationalisierung von Plugins eingehen, um den Nutzen von Fragmenten zu verdeutlichen. Wir beschränken uns in dem Beispiel auf das Ersetzen von Textkonstanten, obwohl es auch durchaus Länderunterschiede in der Anordnung von GUI Elementen usw. haben kann (siehe hierzu Kehn (2002)). Im Wesentlichen sind für uns zwei Fälle möglich.

Textkonstanten im Programm

Hier bietet Eclipse eine Funktion *Externalize Strings* an, mit dessen Hilfe alle verwendeten Stringkonstanten gesucht und angezeigt werden. Es wird dann eine Properties-Datei generiert und die entsprechenden Änderungen am Sourcecode automatisch durchgeführt. So genügt es, die Properties-Datei zu übersetzen.

Textkonstanten in Manifest-Dateien

Bei den Manifest-Dateien muss man diese Properties-Datei selbstständig anlegen. Die Dateien `plugin.properties` etc. können dann übersetzt werden. Die Verknüpfung zwischen der Properties-Datei und dem entsprechenden Eintrag in der Manifest-Datei entsteht, indem man in der Manifest-Datei aus den Stringkonstanten die Leerzeichen entfernt werden und diesen ein %-Zeichen vorangestellt wird (Daum, 2003).

KAPITEL 9

eXtreme Programming - Einführung

Michael Pflug, Daniel Unger

9.1 Einleitung

Ziel dieser Ausarbeitung ist es, extreme Programming (XP) (Beck, 1999) als Programmiermethode vorzustellen, um es dann innerhalb der Projektgruppe 444 einsetzen zu können. Da während der Projektgruppe Eclipse (Eclipse Foundation, 2004) als Entwicklungsumgebung eingesetzt werden soll, wird auf zwei XP-Techniken, die besonders gut mit Hilfe von Eclipse realisierbar sind, genauer eingegangen. Dieses sind die Techniken „Testen“ und „Refactoring“.

Wir beginnen mit einer allgemeinen Einführung in XP, und sagen, warum es im Gegensatz zu konventionellen Techniken erfolgsversprechend erscheint. Anschließend zeigen wir die Funktionsweise von und die Voraussetzungen für den erfolgreichen Einsatz von XP auf. Dann geben wir die Werte an, auf denen XP beruht. Zum Abschluss des ersten Kapitels werden dann die Prinzipien erläutert, die in gewisser Weise als Leitlinien bei der Anwendung von XP eingehalten werden sollten.

Im zweiten Kapitel gehen wir dann detailliert auf die 12 einzelnen XP-Techniken ein, die das eigentliche Herzstück dieses Softwareentwicklungsprozesses bilden. Besonderes Augenmerk wird dabei wie bereits erwähnt auf die Techniken „Refactoring“ und „Testen“ gelegt. Jeweils in einem eigenen Unterkapitel wird der Einsatz der Techniken in Eclipse vorgestellt.

Abschließend wird im dritten Kapitel die Softwareentwicklung mit XP kritisch betrachtet. Hier werden nochmals die positiven Aspekte, wie auch Probleme und Gefahren angeführt. Wir beziehen uns dabei hauptsächlich auf das Buch „Extreme Programming“ (Beck, 2000), das vom „Erfinder“ von XP, Kent Beck, verfasst wurde.

9.2 Allgemeine Einführung in XP

Extreme Programming (XP) ist ein Prozessmodell für die objektorientierte Softwareentwicklung, dass für kleinere Projekte mit unklaren und sich immer wieder ändernden Anforderungen gedacht ist. Es wurde von Kent Beck, Ward Cunningham und Ron Jeffries (Beck, 2000; Beck und Fowler, 2001) entwickelt. Das Programmieren steht hierbei, wie der Name bereits andeutet, im Vordergrund. Die zugrunde liegenden Praktiken sind nicht neu, aber ihre extreme Anwendung unter XP. Mit extremer Anwendung ist die Häufigkeit und Reihenfolge der einzelnen Schritte bei der Softwareentwicklung gemeint. Dabei werden die Schwächen der

einzelnen Praktiken durch die Stärken anderer ausgeglichen. XP ist nach Beck nur dann möglich, wenn alle der vorgestellten 12 Praktiken kombiniert angewendet werden.

In Kapitel 9.2.1 gehen wir darauf ein, welche Probleme bei der herkömmlichen Softwareentwicklung entstehen können und wie XP versucht, sie zu lösen bzw. sie zu vermeiden.

Im darauf folgenden Abschnitt (Kapitel 9.2.2) werden zunächst die Voraussetzungen gezeigt, die nötig sind, damit ein XP-Projekt erfolgreich ist. Anschließend wird grob die Funktionsweise von XP aus der „Vogelperspektive" dargestellt.

Die letzen beiden Kapitel des Abschnittes zeigen zum einen die Werte auf, auf denen XP beruht (Kapitel 9.2.3) und zum anderen die daraus resultierenden Prinzipien, nach denen die Anwender von XP handeln sollen (Kapitel 9.2.4).

9.2.1 Warum XP?

In diesem Abschnitt werden die Grundprobleme der herkömmlichen Softwareentwicklung dargestellt, so dass klar wird, warum überhaupt eine neue Methode entwickelt wurde, bzw. wie XP versucht diese Probleme zu verhindern.

Ein häufiges Problem bei der konventionellen Softwareentwicklung ist, dass es immer wieder zu Terminverzögerungen kommen kann. Diese Verzögerungen werden bei XP in erträglichem Rahmen gehalten, da zuerst die wichtigsten Funktionen implementiert werden, so dass die Funktionen, die zum Liefertermin noch fehlen, von geringerer Bedeutung für den Kunden sind. Zudem sind kurze Releasezyklen von einigen Monaten gefordert.

Falls es bei herkömmlichen Projekten zu solchen Terminverzögerungen kommt, die durch starke Probleme in der Entwicklung bedingt sind, könnte dies zum Projektabbruch führen, ohne dass es ein einziges Release des Programms gegeben hat. XP versucht jetzt diese Probleme zu minimieren, die bei der Entwicklung auftreten können. So wird vom Kunden gefordert aus seinem Anforderungskatalog die Systemmerkmale auszuwählen, die ihm am wichtigsten sind. Diese Funktionen werden als ersten implementiert und dem Kunden als Zwischenreleases zur Verfügung gestellt.

Weiterhin kann sich bei der herkömmlichen Softwareentwicklung ein System als unrentabel herausstellen. Diese ist dann der Fall, wenn die Kosten für Änderungen des Systems oder die Fehlerraten so hoch sind, dass das System ersetzt werden muss. XP versucht dieses Risiko zu minimieren, indem das System in einem solchen Zustand gehalten wird, dass die Fehlerrate relativ gering ist. Dies sollte durch ständiges Testen mit Hilfe automatisierter Komponententests geschehen. Auch führen diese Tests, die nach jeder Änderung ausgeführt werden, dazu, dass die Änderungskosten gering gehalten werden.

Obwohl bei der gebräuchlichen Softwareerstellung natürlich auch getestet wird, kann es vorkommen, dass im Extremfall ein Programm mit hoher Fehlerrate ausgeliefert wird. Um eine Software eben nicht mit relativ hoher Fehlerrate auszuliefern, wird bei XP besonderer Augenmerk auf das Testen während der Entwicklung gelegt. Dabei werden Tests aus zwei Perspektiven geschrieben: Die Programmierer schreiben Tests für die einzelnen Funktionen und der Kunde schreibt Test zur Überprüfung der Leistungsmerkmale des Systems. Dies führt dazu, dass jeder über den funktionierenden Funktionsumfang des Systems informiert bleibt und niemand von einer hohen Fehlerrate überrumpelt wird.

Ein anderes Problem bei der klassischen Vorgehensweise ist, dass die vom Kunden geforderten Leistungsmerkmale falsch verstanden wurden und die Software die Probleme des

Kunden nicht löst. XP versucht dies zu verhindern, indem der Kunde zum Teil des Teams gemacht wird. Ein Kunde ist bei der Programmierung für Fragen ständig zugegen, so dass solche Missverständnisse nicht auftreten können. Ein weiter Vorteil dieser Maßnahme ist es, dass geänderte Wünsche des Kunden in die Entwicklung einfließen können.

9.2.2 Voraussetzungen und Funktionsweise von XP

Um extreme Programming als Methode zur Softwareerstellung innerhalb eines Entwicklerteam erfolgreich einsetzen zu können, bedarf es einiger Vorraussetzungen.

Alle Beteiligten (Programmierer, Management und Kunde) müssen sich auf die genannten Praktiken einlassen. Zum Beispiel muss der Kunde bereit sein, einen Mitarbeiter zur Verfügung zu stellen. Ebenso sollten alle Programmierer am gleichen Ort und zu den gleichen Zeiten arbeiten, da sonst Programmierung in Paaren und Kommunikation erschwert werden. Die Testfälle müssen alle automatisch ausführbar sein und Testdurchläufe sollten nicht zu lange dauern. Weiterhin dürfen die Änderungskosten mit der Zeit nicht so stark ansteigen, da sonst ständiges Refactoring zu teuer wird. Insgesamt ist XP für kleine Teams von 10-15 Entwicklern gedacht, weil die XP-Techniken in größeren Gruppen kaum mehr umzusetzen sind.

An dieser Stelle wollen wir einen kurzen Einblick in die Funktionsweise geben.

Im Gegensatz zu herkömmlichen Entwicklungsprozessen, wie z.B. dem Wasserfallmodell, bei dem die Softwareentwicklung in sequentielle Phasen unterteilt wird, die während der Entwicklung komplett abgeschlossen werden müssen bevor die Arbeit an der nächsten Phase begonnen werden darf (Royce, 1970), verlaufen XP-Projekte in mehreren Iterationen. Die Iterationen werden dabei in einer Art „Spiel“ von den Entwicklern zusammen mit dem Kunden geplant. „Spielregeln“ definieren, wie sich beide Gruppen zu verhalten haben. Im Mittelpunkt stehen dabei rudimentäre Anwendungsfälle, die als „User Storys“ bezeichnet werden. Sie dienen unter anderem als Kommunikationsmittel zwischen Entwicklern und Kunden. Der Kunde schreibt diese Storys auf kleinen Kärtchen. Die Entwickler schätzen den Zeitaufwand für die Implementierung. Wenn der Umfang zu groß ist, schlagen die Entwickler vor, die Story in mehrere einzelne Storys aufzuteilen. Der Kunde priorisiert die Storys und bestimmt gegebenenfalls, welche Storys erst in der nächsten Iteration realisiert werden. Die Storys werden anschließend in kleine Arbeitspakete aufgeteilt, die als Tasks bezeichnet werden.

Die Planungen werden vor Beginn jeder Iteration erneut durchgeführt. Nach jeder Iteration wird bereits ein Release erstellt, das bereits ein funktionierendes System darstellt. Eine Iteration sollte eine Dauer von ein bis drei Wochen haben. Wenn sich herausstellt, dass die Schätzungen der Entwickler daneben liegen, kann auch nachverhandelt werden.

9.2.3 XP Werte

XP beruht auf 4 Werten: Kommunikation, Einfachheit, Feedback und Mut. Diese Werte sollen die genannten Probleme herkömmlicher Softwareentwicklung zuerst auf relativ abstrakte Weise angehen, später werden gezielte XP-Prinzipien daraus entwickelt. Aus diesen Prinzipien entstehen dann letztlich die eigentlichen XP-Techniken.

Kommunikation

Viele Probleme resultieren aus mangelnder Kommunikation („dass jemand etwas Wichtiges nicht mit den anderen bespricht“ (Beck, 2000, Seite 29)). Die mangelnde Kommunikation betrifft sämtliche Bereiche der Entwicklung. Dies können z.B. nicht besprochene Designänderungen, Missverständnisse mit dem Kunden oder falsche Annahmen der Manager über den Projektstatus sein. Diese Art von Kommunikationsdefizit ist durch bestimmte Faktoren bedingt (z.B. die „Bestrafung“ des Programmierers, wenn er dem Manager den wahrhaftigen Projektstatus mitteilt) und demnach änderbar.

Einfachheit

Der zweite Wert, dem XP folgen soll, ist Einfachheit. Das Team muss sich immer wieder die Frage stellen, wie die zu lösenden Probleme am einfachsten angegangen werden können („What is the simplest thing that could possibly work?“ (Beck, 2000, Seite 30)). Dies verlangt, dass man sich nur mit den Dingen beschäftigt, die das momentane Problem lösen. Hierzu muss man jegliche Art von zwanghaftem Vorausdenken abstellen. Dieser Wert resultiert aus der neuen Grundannahme der Änderungskosten: Lieber heute etwas Einfaches tun und falls später doch eine kompliziertere Lösung gebraucht wird, kann man die momentane Lösung immer noch ändern.

Feedback

Zur realistischen Einschätzung des Projektstatus oder um zu sehen, ob die Funktion, die man gerade implementiert hat, auch funktioniert, braucht man direktes und stetiges Feedback. Feedback ergänzt die bereits genannten Werte. Je mehr Feedback man bekommt, umso einfacher und ehrlicher ist Kommunikation.

Mut

Ein weiterer Wert von XP ist Mut. Dieser ist in mehrerlei Hinsicht gefordert. Zum einen muss man sich immer gegen die altbewährten Methoden der Softwareentwicklung durchsetzen. Zum anderen muss man mutig an Änderungen herangehen, auch wenn man das betroffene Modul nicht selber geschrieben hat. Auch die Einfachheit des Systems fordert Mut. Etwas so zu entwerfen, dass es nur für die momentanen Anforderungen ausreicht, widerspricht den gängigen Methoden, die bspw. an den Universitäten den Entwicklern beigebracht, Systeme flexibel und erweiterbar zu planen. Ferner muss man so mutig sein, Code an dem man lange gearbeitet und „experimentiert“ hat, wegzuwerfen und mit einem neuen Lösungsansatz zu beginnen.

9.2.4 XP Prinzipien

In diesem Abschnitt werden den vier grundsätzlichen Werten von XP bestimmte Prinzipien zugeordnet, die als Entscheidungshilfe dienen. Jedes Prinzip verkörpert die XP-Werte. Werte sind etwas unbestimmtes, die einem nicht helfen, konkret zu handeln. Deshalb werden im folgenden fünf zentrale Prinzipien aufgeführt, die dieses konkrete Handeln ermöglichen. Neben diesen 5 zentralen Prinzipien gibt es noch eine Reihe weniger wichtiger Merkmale. Diese haben aber nicht so einen zentralen Charakter wie die im folgenden Erwähnten.

Unmittelbares Feedback

Wichtig ist direktes, unmittelbares Feedback: Sowohl direktes Feedback vom Kunden (je kürzer die zeitliche Differenz, desto größer der Lernerfolg), ermöglicht durch kurze Releasezyklen, als auch direktes Feedback vom System, ermöglicht durch Komponententests.

Einfachheit anstreben

Es ist lediglich die heute aktuelle Aufgabe zu erledigen und das so einfach wie möglich, d.h. ohne auf zukünftige Problemstellungen einzugehen. Die Module/Funktionen, bei denen später festgestellt wird, dass sie komplexer sein müssen, kann man dann anschließend in der Zeit erledigen, die bei der Programmierung der anderen Funktionen gespart wurde.

Inkrementelle Veränderungen

Jede Problemstellung wird in viele kleine Problemstellungen zerlegt. Dies vereinfacht die Programmierung und das Testen der Funktionen, und man bewahrt sich vor umfassenden Veränderungen.

Veränderungen wollen

Man darf nicht schlecht auf Veränderungen vorbereitet sein. Veränderungen erwarten, reicht aber auch nicht. Man muss Veränderungen wollen, da man weiß, dass diese keinen negativen Effekt haben, bzw. man sie sogar positiv für sich nutzen kann. Die beste Alternative einer Entscheidung ist demnach immer die, die die meisten Optionen offen hält.

Qualitätsarbeit

Qualität darf unter XP Gesichtspunkten nicht als freie Variable betrachtet werden (eigentlich sonst auch nicht). Denn jeder möchte gute Arbeit abliefern. Andernfalls sind die Teammitglieder schnell frustriert, und das Projekt scheitert.

9.3 Die 12 XP-Techniken

Mit Techniken werden bei XP die Verfahren bezeichnet, mit denen unter XP gearbeitet wird. Man spricht in diesem Zusammenhang auch von den „12 Säulen des XP“. Diese Verfahren werden in den folgenden Unterkapiteln nun vorgestellt. Dabei beziehen wir uns weitestgehend auf (Beck, 2000), nur bei den Kapiteln über Testen und Refactoring erweitern wir die Ausführungen auf den Gebrauch dieser Techniken in der Programmierumgebung Eclipse. Hierbei ist es laut Beck wichtig zu erkennen, dass die einzelnen XP-Techniken alleine keinen Vorteil bringen, sondern nur im Zusammenspiel die gewünschten Erfolge erzielen.

9.3.1 Das Planungsspiel

Das Planungsspiel vereint das Festlegen des Umfangs der nächsten Programmversion mit den Aufwandsschätzungen der Programmierer. Es ist also die Planungsphase von XP. Der Plan wird hierbei ständig aktualisiert. Dabei hat die Geschäftsseite folgende Entscheidungen zu treffen:

- Der Umfang der Software wird festgelegt. Dabei wird darauf geachtet, dass weder zu wenig noch zu viel einbezogen wird.
- Den einzelnen Funktionen müssen Prioritäten zugeordnet werden. Welche Funktionen sollen direkt implementiert werden, welche sind nicht so wichtig?
- Die Zusammensetzung von Versionen muss bestimmt werden. Wann macht ein erstes Release von Geschäftsseite Sinn? Welche Funktionen müssen dann bis zum nächsten Release implementiert werden?
- Letztendlich müssen Liefertermine festgelegt werden.

Die Programmierer unterstützen die Geschäftsseite dabei, indem sie folgende Entscheidungen treffen:

- Schätzung des Aufwands zur Implementierung jeder einzelnen Funktion
- Technische Konsequenzen bei Entscheidungen der Geschäftsseite für eine bestimmte Alternative, beispielsweise für eine bestimmte Datenbank.
- Strukturierung der Teamarbeit bezogen auf die Umgebung.
- Genaue Terminplanung für die Implementierung der Funktionen nach den Prioritäten der Geschäftsseite.

9.3.2 Kurze Releasezyklen

XP fordert kurze Releaszyklen von einigen Monaten, um auf Feedback aus dem Kundenbetrieb zurückgreifen zu können. Dabei sollte man aber nur vollständig implementierte Releases ausgeliefert werden.

9.3.3 Metapher

Die Metapher ersetzt bei XP das, was sonst als Architektur bezeichnet wird. Alle technischen Komponenten sollten in dieser Metapher beschrieben werden, so dass man durch sie die Funktion der Software leichter verstehen kann. Dabei werden die Komponenten des Systems mit Begriffen aus der realen Welt beschrieben. Letztlich entwickelt man so eine gemeinsame Sprache/Vokabular, um über die Arbeitsweisen und das zu erstellende System diskutieren zu können.

9.3.4 Einfaches Design

Das Design sollte möglichst einfach gehalten werden. Ein einfaches Design erfüllt dabei folgende Eigenschaften:

- Besteht alle Tests
- Hat keine Redundanzen

- Selbsterklärender Code
- Geringst mögliche Anzahl von Klassen und Methoden.

Aufgrund der geringen Änderungskosten sollte man nur das implementieren, was für die derzeitige Funktionalität nötig ist. Das Design erfüllt also zu jedem Zeitpunkt die genannten Kriterien, ohne dass auf zukünftige Arbeiten Rücksicht genommen wird.

9.3.5 Testen

Das Testen ist eine der Kerneigenschaften von XP. Für jede logische Programmeigenschaft existieren Tests. Es darf erst weiter gearbeitet werden, wenn alle Testfälle erfolgreich waren. Die Tests sollten, realisiert durch bestimmte Testumgebungen, automatisch ausführbar sein. Getestet wird durch Komponententests der Programmierer und durch Funktionstests des Kunden.

Wichtig ist in diesem Zusammenhang auch die Reihenfolge des Programmierens von Tests und Programmcode. Bei XP wird zuerst der Test für eine Funktion geschrieben. Dann fängt man an, die Funktion zu programmieren. Dieses wird als testgetriebenes Programmieren bezeichnet. Sobald alle Tests bestanden sind, ist die Funktion fertig (Westphal, 2000).

Testen mittels JUnit in Eclipse

Eclipse bietet in seinen aktuellen Versionen ein integriertes Framework, dass für die testgetriebenen Programmierung bestens geeignet ist. JUnit (Beck und Gamma, 2001), so der Name diese Frameworks, ist eine automatisierte Testumgebung, die von Kent Beck und Erich Gamma entwickelt wurde. Für eine gute Einführung eignet sich das Buch (Link, 2002). JUnit erlaubt es dem Programmierer, auf einfache Art und Weise, Tests für seine Klassen und Methoden zu schreiben und sie anschließend alle auszuführen.

Das Vorgehen bei JUnit ist folgendes: Man erzeugt pro Klasse, die man implementieren will, eine Testklasse die von `TestCase` (Klasse aus dem JUnit-Framework) abgeleitet wird. Wie es beim testgetriebenen Programmieren vorgeschrieben ist, wird für jede Eigenschaft, die implementiert werden soll, zunächst eine Testmethode (ein Testfall) geschrieben, die diese Eigenschaft überprüft. Anhand der graphischen Oberfläche von JUnit kann dann direkt erkannt werden, ob die Tests erfolgreich waren oder nicht (ein Balken wird entweder grün oder rot). Falls sich der Balken rot färbt, ist ein Fehler aufgetreten, der dann anhand eines Call-Stacks identifiziert werden kann. Der Fehler wird behoben oder aber der Testfall angepasst.

9.3.6 Refactoring

Refactoring bedeutet, dass ständig versucht wird, das Programm zu verbessern und zu vereinfachen. Das hat zur Folge, dass manchmal mehr Arbeit nötig ist, als wenn die eine alte Version, die auch alle Tests besteht, erhalten bleibt. Es wird aber davon ausgegangen, dass sich diese Mehrarbeit mittel- bzw. langfristig auszahlt, denn sie ist wesentlicher Bestandteil, um das System änderungsfähig zu halten und geringe Änderungskosten zu ermöglichen.

Ein weiterer mit der Änderbarkeit zusammenhängender Teil des Refactorings besteht in der Dokumentation. Es wurde festgestellt, dass mit verstreichender Projektlaufzeit die Do-

kumentation immer weniger angepasst wird (Lippert u. a., 2002). XP beschränkt daher die Dokumentation auf den Code. Dieser sollte in hohem Maße selbsterklärend sein.

Refactoring mit Eclipse

Eine Entwicklungsumgebung mit Refactoringbrowser, wie in unserem Fall Eclipse, kann diese Arbeit erleichtern und viele Refaktorisierungen aus dem Katalog per Mausklick erledigen. So lassen sich sehr leicht Variablen umbenennen, Methoden extrahieren, in andere Klassen verschieben und vieles mehr.

In Eclipse werden folgende Refactorisierungsmöglichkeiten angeboten (Schiffer und Violka, 2003): Umbenennen, Verschieben, Methodensignatur verändern, Konvertieren von einer anonymen Klasse in eine verschachtelte Klasse, Verschieben von Methoden in Ober- und Unterklassen, Umwandeln von Klassen in Interfaces usw.

9.3.7 Programmieren in Paaren

Eines der auffälligsten Merkmale von XP, das es von anderen Entwicklungsprozessen unterscheidet, ist, dass immer zwei Programmierer zusammen an einem Rechner arbeiten sollen. Der Entwickler, der Maus und Tastatur bedient, macht sich Gedanken über die zurzeit zu implementierende Funktion. Der andere überlegt, ob der Ansatz zur Implementierung überhaupt funktioniert, macht sich Gedanken über Testfälle, die scheitern könnten und ob es Möglichkeiten zum Refactoring gibt. Dabei tauschen die beiden Entwickler regelmäßig in kurzen Zeitabständen die Rollen. Auf diese Weise werden der Code, der Entwurf und die Tests einer kontinuierlichen Begutachtung unterzogen. Die Programmiererpaare sind hierbei nicht unveränderlich. Diese Wechsel unterstützen, wie das gesamte Programmieren in Paaren, den Grundwert der Kommunikation im Team. Die dynamische Paarbildung hat den positiven Nebeneffekt, dass sich das Wissen über alle Aspekte des Systems auf alle Entwickler verbreitet.

9.3.8 Gemeinsame Verantwortlichkeit

Der Code gilt unter XP nicht als Privateigentum desjenigen, der ihn programmiert hat, sondern ist Gemeineigentum. Bei der herkömmlichen Art Software zu entwickeln, wurde der Code meist nur von dem geändert, der ihn auch selber geschrieben hat. Wollte man dennoch diesen Code selber ändern, geschah dieses meistens nur mit Absprache desjenigen, der den Code erstellt hat. Folglich fand Refactoring auch nur nach solcher Absprache statt. Unter XP wird von jedem Programmierer erwartet, dass er, wenn er eine Verbesserungsmöglichkeit sieht, diese sofort durchführt. Er kann ja unmittelbar sehen, ob noch alle Tests funktionieren. Durch die „gemeinsame Verantwortlichkeit“ wird verhindert, dass Code aus dem Wissensstand eines einzigen Programmierers entsteht.

9.3.9 Fortlaufende Integration

Der neue Code wird ständig integriert und getestet, spätestens jedoch nach einem Arbeitstag. Dafür wird ein eigener Integrationsrechner bereitgestellt. Nach der Programmierarbeit geht man an diesen Integrationsrechner, integriert und testet bis alle Tests fehlerfrei sind. Generell sollte es so sein, dass bereits nach einigen wenigen Änderungen die eigenen Arbeit auf den

Integrationsrechner gespielt werden. So vermeidet man Konflikte beim Zusammenführen und macht die gerade neu entwickelten Funktionen dem gesamten Team zugänglich.

9.3.10 40-Stunden Woche

Es sollte maximal eine Woche Überstunden gemacht werden. Sollten mehr Überstunden von Nöten sein, muss die Ursache dafür gesucht werden und/oder gegebenenfalls der Projektplan angepasst werden. Zudem sollte jeder Programmierer mindestens zwei Wochen im Jahr Urlaub am Stück nehmen. Nur unter dieser Voraussetzung sind die Programmierer ausgeruht und leisten gute Arbeit.

9.3.11 Kunde vor Ort

Ein Kunde, der das System später verwendet, sollte bei der Entwicklung anwesend oder mindestens ständig erreichbar sein und so für eventuelle Fragen zur Verfügung stehen. Nur so ist bei der Entwicklung direktes Feedback durch den Anwender realisierbar, und Missverständnisse über Umfang und Funktionen des Programms werden verhindert.

9.3.12 Programmierstandards

Gemeinsame Programmierstandards sind deswegen nötig, da der Code ständig von allen Programmierern verstanden, überarbeitet oder ergänzt werden muss. Der Standard wird am Anfang einheitlich festgelegt und alle sollten sich freiwillig an diesen halten. Auch dient ein gemeinsamer Standard dazu, dass man nach der Fertigstellung einer Funktion nicht erkennen können sollte, wer diese geschrieben hat.

9.4 Fazit

Nachdem wir XP nun soweit eingeführt haben, dass man es als Methode zur Softwareentwicklung einsetzen kann, zeigen wir abschließend nochmals die Vorteile (Kapitel 9.4.1), sowie mit XP verbundene Probleme und ev. Gefahren auf (Kapitel 9.4.2), die beim der Nutzung auftreten können. Denn auch wenn sich diese Form des Software-Engeneering in einigen Punkten verlockend anhört, ist sie doch an den ein oder anderen Stellen kritische zu hinterfragen.

9.4.1 XP Vorteile

Ein bedeutender Vorteil von XP ist es, dass während der gesamten Entwicklung ein funktionierendes und getestetes System existiert, dass dem Kunden zur Verfügung steht. Er kann und soll direkt in den Entwicklungsprozess eingreifen und muss nicht auf ein fertig ausgeliefertes Produkt warten. So hat der Kunde beispielsweise die Möglichkeit, Features nachträglich in das Planungsspiel mit einzubringen, die er im Laufe der Entwicklung für nützlich erachtet. Zusätzlich kann er diese mit Prioritäten versehen, so dass die wichtigsten Eigenschaften auch zuerst realisiert werden.

Durch ständiges Refaktorisieren des Quellcodes wird dieser mit der Zeit nicht unleserlicher, sondern bleibt auch bei zukünftigen Änderungen einfach und selbsterklärend. Somit können sich auch neue Teammitglieder schnell in die bestehende Codestruktur einlesen.

Die Berichte der Entwickler, die XP eingesetzt haben, zeigen hohe Zufriedenheit bei allen Beteiligten. Programmierer berichten von Motivation und Freude bei der Arbeit, das Management freut sich über die gute Termineinhaltung, und die Kunden begrüßen die frühe Verfügbarkeit eines funktionierenden Systems sowie die hohe Qualität (Lippert u. a., 2002).

Dadurch, dass ein enges Zusammenspiel zwischen Entwickler und Kunden gegeben ist, wird ein Produkt entwickelt, das den Wünschen des Kunden eher entspricht, als mit herkömmlichen Verfahren. Der Kunde bringt das nötige Fachwissen mit, welches fachfremde Entwickler meist nicht besitzen.

Nachteile einer Technik werden durch Vorteile anderer Techniken ausgeglichen. So wird z.B. das Fehlen eines Anforderungsdokuments dadurch ausgeglichen, dass der Kunde jederzeit präsent ist.

9.4.2 XP Nachteile

Ein Problem ist das Fehlen von Dokumentation. Es gibt lediglich die Dokumentation in Form der Testfälle und des Codes. Dies ist zwar für die Mitglieder des XP-Teams meistens ausreichend, wenn allerdings später jemand etwas ändern soll, der nicht in das Projekt involviert war, könnten Probleme auftreten.

Die XP Praktiken setzen funktionierende Tests voraus. Durch das ständige Refactoring müssen aber auch diese Tests angepasst werden, so dass es vorkommen kann, dass die Tests das Falsche testen. Die einzige Kontrolle, ob die Tests richtig geschrieben wurden, erfolgt einzig durch den Programmierpartner beim Programmieren in Paaren.

Auch das gemeinsame Code-Eigentum kann Probleme verursachen. Durch das Fehlen von Dokumentation müssen die Programmierer das Design auswendig kennen. Da es aber ständig geändert wird (Refactoring), ist dies fast unmöglich, so dass unter falschen Voraussetzungen codiert wird. Außerdem kann es zu konkurrierenden Änderungen an denselben Klassen kommen, was Integrationsproblemen nach sich zieht. Dies muss aber kein reines XP Problem sein, sondern kann auch ein Zeichen von allgemeinen Kommunikationsproblemen der Entwickler sein.

Ferner ist XP bis jetzt unzureichend dokumentiert. Beispielsweise wird kaum beschrieben, wie die Systemmetapher gewählt werden sollte und wie diese auszusehen hat. Außerdem gibt es keine Daten, die nachweisen, dass XP anderen Vorgehensweisen überlegen ist. Uns sind jedenfalls keine empirischen Untersuchungen bekannt, die etwas über den Erfolg von XP aussagen.

XP funktioniert nach Beck nur dann, wenn alle Techniken während der Entwicklung kombiniert angewendet werden. Falls auch nur eine Technik nicht zum Einsatz kommt, würde der XP- Entwicklungsprozess wie ein Kartenhaus in sich zusammenfallen, d.h. die Softwareentwicklung wäre dann nicht mehr so effektiv, wie erwartet. Alle Techniken sollten sich also zu einem vollständigen Ganzen ergänzen.

KAPITEL 10

Entwurfsmuster

Michél Kersjes, René Schönlein

10.1 Was ist ein Entwurfsmuster?

Entwurfsmuster sind auf Erfahrungen beruhende abstrakte Lösungsbeschreibungen für allgemeine Probleme in der objektorientierten Softwareentwicklung. Sie werden nicht entwickelt, vielmehr werden sie gefunden und stellen dokumentiertes Expertenwissen dar. Die Lösungsbeschreibung wird durch ihre Komponenten, den Beziehungen zwischen ihnen, und ihr Zusammenspiel spezifiziert. Die Muster werden stets im Entwurf verwendet und sind sprachunabhängig.

10.1.1 Die Geschichte der Entwurfsmuster

Der Begriff des Musters wurde 1977 von Christoper Alexander in seinem Werk „A Pattern Language“ geprägt:

> „Ein Muster beschreibt ein in unserer Umwelt beständig wiederkehrendes Problem und erläutert den Kern der Lösung für dieses Problem, so dass man diese Lösung beliebig oft anwenden kann, ohne sie zweimal gleich auszuführen.“ (Alexander u. a., 1977)

Christopher Alexander bezieht sich mit seiner Aussage auf Muster in Gebäuden und Städten. Dennoch trifft seine Aussage ebenfalls auf Entwurfsmuster im Softwaredesign zu. Ward Cunningham und Kent Beck greifen im Jahr 1987 die Ideen von Alexander für die Softwareentwicklung auf. Jahre bevor die objektorientierte Programmierung weit verbreitet war, entwickelten diese beiden – inspiriert durch Alexander – eine kleine Mustersprache bestehend aus fünf zusammenhängenden Mustern. Etwa zur gleichen Zeit beginnt Jim Coplien mit der Sammlung und Zusammenstellung von Idiomen für sein Buch „Advanced C++ Programming Sytles and Idioms“, welches 1991 erscheint. Im Sommer 1993 formiert sich dann die „Hillside Generative Patterns Group“, die 1994 die erste Entwurfsmusterkonferenz mit dem Namen „Pattern Language of Programming“ (PLoP) organisiert (The Hillside Group). Im selben Jahr veröffentlicht eine Gruppe von vier Leuten, die so genannte „Gang of Four“ (Gamma, Helm, Johnson, Vlissides), das wohl bekannteste Buch über Entwurfsmuster, welches auch heute noch eines der bedeutendsten Werke zu Entwurfsmustern darstellt. Es trägt den Titel „Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software“ (Gamma u. a.,

2001) und stellt einen gut strukturierten Entwurfsmusterkatalog vor. Mit diesem Werk halten Muster auf breiter Front Einzug in die Informatik.

10.1.2 Warum beschäftigt man sich mit Entwurfsmustern?

Entwurfsmuster vereinfachen die Wiederverwendung von guten Entwürfen und verhindern, dass man eine Lösung für ein Problem sucht, welches bereits häufig von erfahrenen Leuten gelöst wurde. So ist es auch Entwicklern mit weniger Erfahrung möglich, bessere Entwürfe zu erstellen. Des Weiteren steht Entwicklern so ein geeignetes Vokabular zur Verfügung, um sich über Probleme auszutauschen.

10.1.3 Beschreibung von Entwurfsmustern

Wir wollen Entwurfsmuster in einem einheitlichen Format beschreiben, um sie leichter miteinander vergleichen und intuitiver verstehen zu können. Daher werden wir die Muster anhand des folgenden Schemas, inspiriert durch die „Gang of Four“ (Gamma u. a., 2001), erläutern:

Mustername : Soll auf prägnante Weise das Verhalten des Musters beschreiben.
Zweck : Kurze Darstellung der Intention des Musters.
Motivation : Darstellung eines Problems, zu dem das entsprechende Muster einen Lösungsansatz bietet.
Anwendbarkeit : Welche konkreten Gegebenheiten müssen erfüllt sein, um das Muster sinnvoll anwenden zu können?
Technische Realisierung : Zeigt ein Strukturdiagramm des entsprechenden Musters und beschreibt die einzelnen Komponenten des Diagramms sowie die Interaktionen zwischen den Komponenten.
Konsequenzen : Erläuterung der Vor- und Nachteile des Musters.

10.1.4 Klassifizierung von Entwurfsmustern

Die meisten Entwurfsmuster lassen sich in drei Unterarten aufgliedern, je nachdem welchen Zweck sie verfolgen.

Erzeugungsmuster : Betreffen den Prozess der Objekterzeugung
Strukturmuster : Behandeln die Zusammensetzung von Klassen und Objekten
Verhaltensmuster : Erläutern, wie Klassen und Objekte Zuständigkeiten und die Zusammenarbeit aufteilen

Diese Einteilung lässt sich aber noch verfeinern, indem wir die Entwurfsmuster noch dahingehend einordnen, auf welchen strukturellen Bereich sie sich hauptsächlich konzentrieren.

Klassenbasiert : Beziehen sich auf Klassen; Unterklassenbildung durch Vererbung
Objektbasiert : Beziehen sich auf Objekte; Objektkomposition durch Delegation

10.1.5 Wie findet man das richtige Entwurfsmuster?

Das richtige Entwurfsmuster für ein Problem zu finden gestaltet sich aufgrund der Vielzahl dieser Muster recht schwer, vor allem wenn die Materie für den Anwender noch neu ist. Erschwerend kommt hinzu, dass es keinen optimalen Weg gibt, genau das richtige Muster herauszufinden. Daher bieten sich einige unterschiedliche Herangehensweisen an das Problem an.

Einerseits könnte man sich überlegen, wie Entwurfsmuster die ihnen gestellten Probleme lösen. So gibt es einige grundlegende Techniken, die von vielen Mustern genutzt werden. Zu nennen wären da Vererbung, Komposition und Delegation. Wenn man diese versteht, kann man sich das ungefähre Aussehen eines geeigneten Entwurfsmusters selber erarbeiten. Im Anschluss daran kann man dann in einem Katalog von Entwurfsmustern eine geeignete Lösung heraussuchen und deren Konsequenzen vergleichen.

Eine andere Möglichkeit ist es, sich einfach die kurzen Intentionen aller Entwurfsmuster anzuschauen. Auf diesem Wege lassen sich schnell interessant klingende Vorschläge heraussuchen. Wenn man auch so nicht auf ein geeignetes Muster stößt, kann man versuchen, bei einem artverwandten Entwurfsmuster eine Lösung für das Problem zu finden.

Ferner ist es natürlich möglich, sich die Gründe für ein Neudesign anzuschauen und diejenigen herauszusuchen, die eventuell beim eigenen Projekt auftreten könnten. Mit diesem Wissen lassen sich dann die Entwurfsmuster herausfinden, die das Design so flexibel halten, dass ein Neudesign vermieden werden kann.

Eine gegensätzliche Vorgehensweise dazu ist, dass man sich bereits vor dem Design Gedanken macht, was variabel bleiben soll. Anhand dieser Vorgaben lassen sich dann ebenfalls geeignete Entwurfsmuster herausfinden.

10.2 Ein einführendes Beispiel

Anhand des Model-View-Controller-Konzeptes (MVC) wollen wir nun zeigen, wie Muster in der Praxis vorkommen können. Das MVC-Konzept findet vor allem bei grafischen Benutzeroberflächen (GUI) Anwendung und wurde in Smalltalk-80 erstmals zur Konstruktion von Benutzungsschnittstellen verwendet. MVC entkoppelt das traditionelle Eingabe-Verarbeitung-Ausgabe-Prinzip. Wie der Name schon andeutet, besteht MVC aus drei Objektarten. Das **Model-Objekt**, welches den verarbeitenden und verwaltenden Teil der Anwendung darstellt, das **View-Objekt**, das sich um die visuelle Repräsentation der Daten kümmert, und das **Controller-Objekt**, welches für die Verarbeitung sämtlicher Eingaben zuständig ist.

10.2.1 MVC Interaktion

Das Model-Objekt stellt den internen Zustand des Systems dar. Dieser Zustand wird von einem View-Objekt präsentiert. Es können jedoch auch mehrere verschiedene View-Objekte den Zustand wiedergeben, wie Abbildung 1 zeigt. Daher muss das Model-Objekt die View-Objekte benachrichtigen, sobald sich an seinem Zustand etwas ändert, so dass die View-Objekte sich die aktuellen Daten holen und präsentieren

Die dritte Komponente, das Controller-Objekt, nimmt Eingaben entgegen. Diese Eingaben

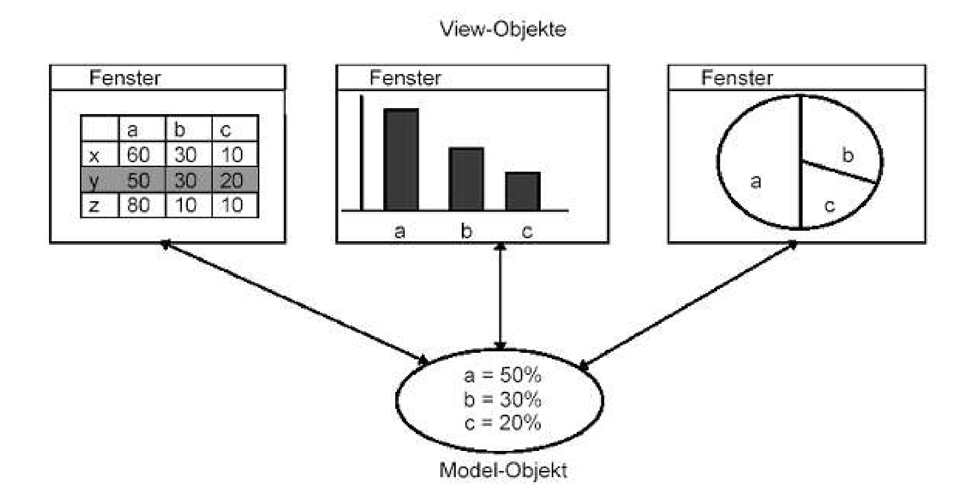


Abbildung 10.1.: Beziehung zwischen Model und View-Objekten (Gamma u. a., 2001)

ändern den Zustand des Model-Objekts und somit indirekt auch die Darstellung. Die Unterscheidung zwischen View und Controller fällt mitunter recht schwer, da Views ebenfalls Eingaben erlauben können (siehe z.B. Abbildung 10.1, linker View).

10.2.2 Entwurfsmuster in MVC

Im MVC-Konzept kann man mehrere Entwurfsmuster wieder finden. Von diesen wollen wir nun zwei exemplarisch vorstellen. Die Beziehung zwischen der Model-Komponente und den verschiedenen Sichten von MVC ist ein Beispiel für das Beobachtermuster. Dies ist genau deshalb der Fall, da es eine Registrierungs- und Benachrichtigungsinteraktion zwischen diesen beiden Komponenten gibt. Sollte sich etwas am Zustand des Modells ändern, werden die Registrierten Sichten darauf hingewiesen und versorgen sich mit den nötigen Daten aus dem Modell, um ihre Darstellung den neuen Gegebenheiten anzupassen.

Die Model-Komponente verfügt über eine Schnittstelle die es ermöglicht Beobachterobjekte an- und abzumelden. Es spielt hier keine Rolle wie hoch die Anzahl der Beobachterobjekte ist. Hier wird auch der Updateprozess angestoßen, indem eine Nachricht an alle angemeldeten Beobachter gesendet wird. Die Views implementieren eine Operation zum Abgleich der Daten mit dem Model. Sie halten sich eine Referenz auf das von ihnen beobachtete Objekt, um bei dem Updateprozess ihrer Darstellung, auf die im Model gespeicherten Daten zugreifen zu können.

Zunächst melden sich die Objekte der beiden Komponenten Controller und View am Model an. Der Controller ruft die Methode „*setzeZustand*“ des von ihm beobachteten Modells auf. Hierdurch werden die Daten des Modells verändert, so dass alle registrierten Views über

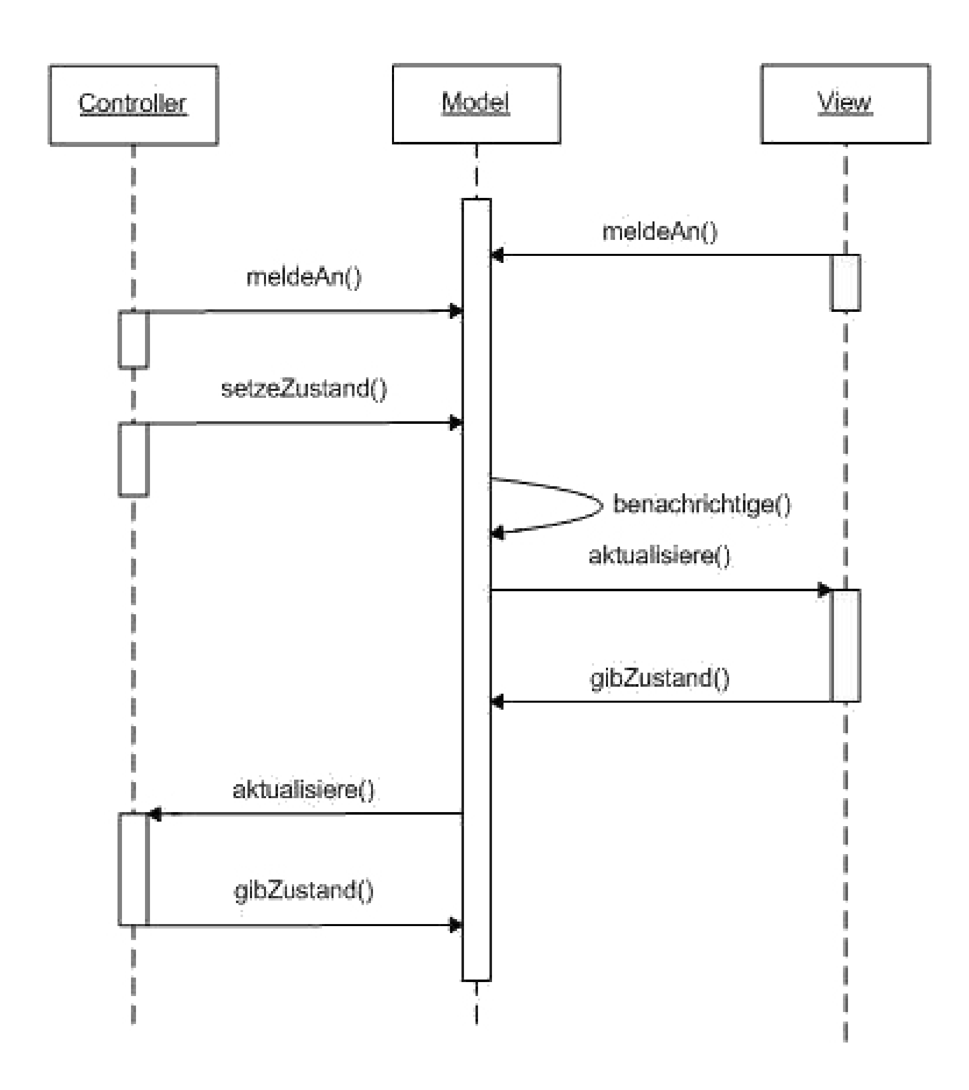


Abbildung 10.2.: Ablauf der Interaktion zwischen den 3 MVC Komponenten (Gamma u. a., 2001)

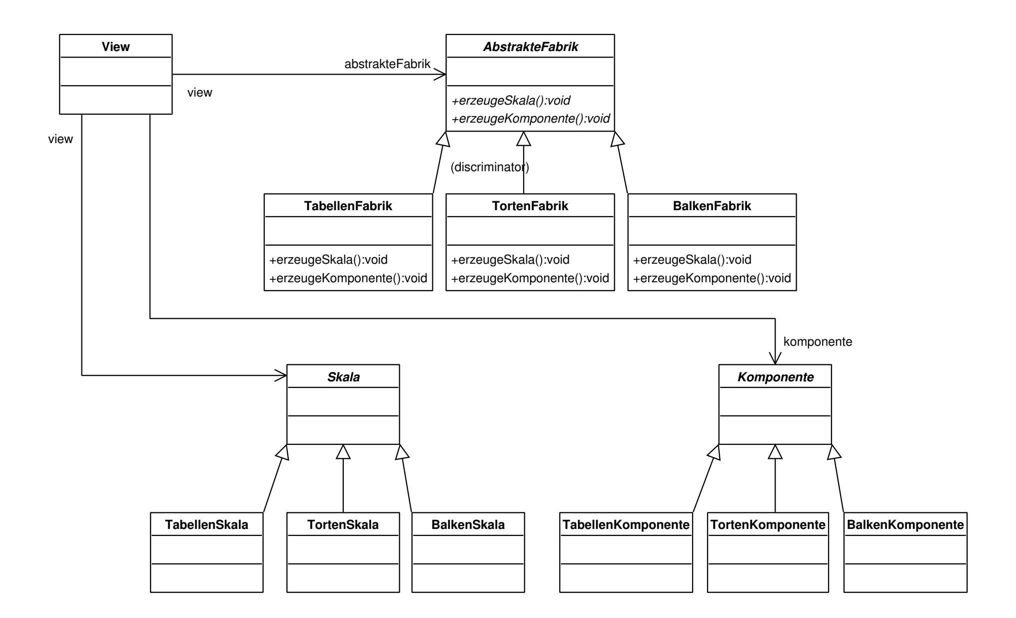


Abbildung 10.3.: Struktur der Abstrakten Fabrik am Beispiel

diese Änderung zu informieren sind. Das Modell benachrichtigt daraufhin alle registrierten Views. Die dadurch angestoßenen Views holen sich anschließend mit Hilfe der Methode „*gibZustand*“ den neuen Zustand des Modells und ändern ihre Darstellung.

Das zweite Entwurfsmuster, welches man in MVC finden kann, ist die Abstrakte Fabrik. Views können die Daten des Models auf unterschiedliche Weise graphisch aufbereiten. Man könnte nun die Art, wie ein konkreter View die für ihn interessanten Daten darstellt, komplett dem View überlassen. Dieses Vorgehen hat nur den großen Nachteil, dass man für jede unterschiedliche Darstellungsart eine eigene View-Klasse implementieren muss. Hier kann uns das Abstrakte Fabrik Muster nun dabei helfen, diesen Nachteil zu umgehen, indem es eine Klasse einführt, die für die Erzeugung der einzelnen Darstellungskomponenten verantwortlich ist. Die Views können nun auf die Erzeugung der für sie konkreten Darstellungskomponenten verzichten, sondern bemühen einfach eine passende Fabrikklasse mit dieser Aufgabe.

Dazu wird eine abstrakte Klasse (*abstrakteFabrik*) eingeführt, die die Schnittstelle für alle konkreten Fabriken definiert. Die konkreten Fabriken implementieren diese Schnittstelle und sind für die Erzeugung der entsprechenden Elemente zuständig. Die konkrete Fabrik (*BalkenFabrik*) ist somit für die Erzeugung der Komponenten eines Balkendiagramms zuständig, also für die im Diagramm aufgezeigten Objekte von „*BalkenSkala*“ und „*BalkenKomponente*“ (siehe Abbildung 10.3).

Durch die einheitliche Schnittstelle ist es nicht mehr nötig für jede andersartige View eine spezifische Implementierung der Klasse View zu haben. Es genügt eine einzige Implementierung der Klasse View, die lediglich verschiedene Fabriken (z.B. *BalkenFabrik*, *TortenFabrik*)

für die gewünschte Darstellung nutzt. In der Klasse View könnte z.B. eine Operation „*zeichne-Komponente*“ aufgerufen werden, die dann eine konkrete Fabrik dazu auffordert seine Komponente zu zeichnen.

10.3 Strukturmuster

„Strukturmuster befassen sich mit der Komposition von Klassen und Objekten, um größere Strukturen zu bilden“ (Gamma u. a., 2001). Welche Beweggründe hierfür existieren, lassen sich allerdings nicht einheitlich zusammenfassen. Es gibt Entwurfsmuster die sich direkt mit dem Zusammensetzten von Objekten zu komplexen Strukturen befassen, oder dem Benutzer ein Objekt als Stellvertreter anbieten, welches das „reale Objekt“ simuliert. Das Adaptermuster, welches wir im weiteren Verlauf noch kennen lernen werden, passt z.B. eine Schnittstelle einer Klasse an eine andere an und bietet somit eine Abstraktion, durch die man dann erreicht, dass auf beiden Klassen gearbeitet werden kann.

Dieser Abschnitt soll keinen umfassenden Überblick über die Strukturmuster geben, sondern vielmehr klar machen, mit was für unterschiedlichen Zielsetzungen Strukturmuster verwendet werden können. Allen Strukturmustern gemeinsam ist die Methodik, um spezifische Ziele zu erreichen. So arbeiten objektbasierte Strukturmuster meist mit Komposition, um Objekte zusammenzuführen und dadurch neue Funktionalität zu gewinnen. Hierdurch erreicht man meist eine hohe Flexibilität, da die komponierten Objekte jeder Zeit, also auch während der Laufzeit, austauschbar sind. Klassenbasierte Strukturmuster hingegen benutzen Vererbung, um Schnittstellen und Implementierungen zusammenzuführen. In unserem Fall spielen die klassenbasierten Muster keine allzu große Rolle. Deshalb werden wir diese nicht weiter ausführen.

10.3.1 Kompositum

Zweck

Fügt Objekte zu Baumstrukturen zusammen, um Teil-Ganzes-Hierarchien zu repräsentieren. Das Kompositionsmuster ermöglicht es Klienten, sowohl einzelne Objekte als auch Kompositionen von Objekten einheitlich zu behandeln.

Motivation

Angenommen, man hat eine Menge von simplen Objekten, die zusammengesetzt ein sinnvolles Objekt ergeben, dann müsste man beim Zugriff auf diese Objekte zwischen simplen und zusammengesetzten Objekten unterscheiden.

Um diese Unterscheidung zu vermeiden, und den Zugriff zu vereinheitlichen, schlägt das Kompositionsmuster eine uniforme Repräsentation vor. Realisiert wird dies durch eine abstrakte Klasse, welche sowohl simple Objekte als auch ihre Behälter repräsentiert.

Anwendbarkeit

Das Kompositionsmuster verwendet man, wenn man Teil-Ganzes-Hierarchien von Objekten darstellen möchte. Man sollte es ebenfalls verwenden, wenn man den Zugriff auf primitive und

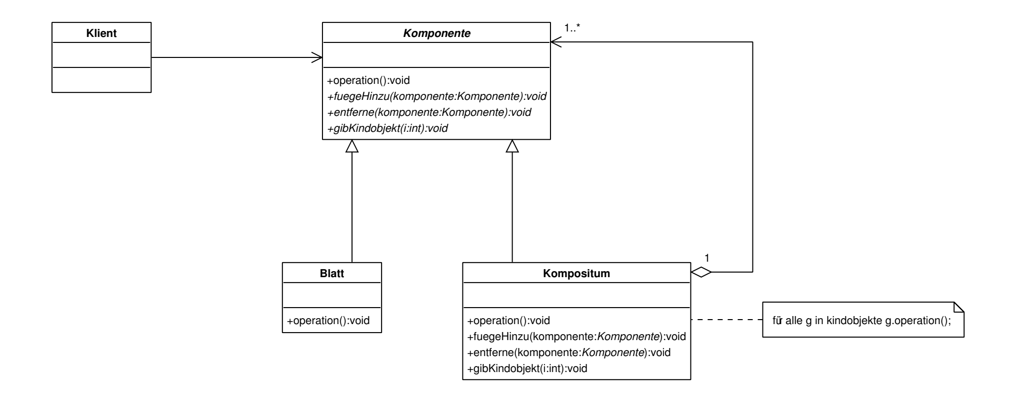


Abbildung 10.4.: Kompositumstruktur

zusammengesetzte Objekte vereinheitlichen möchte, so dass der Klient nicht mehr zwischen diesen unterscheiden muss.

Technische Realisierung

Der Klient greift ausschließlich auf die abstrakte Klasse *Komponente* zu. Diese definiert die Schnittstelle, welche den Zugriff auf die Kindobjekte ermöglicht, wobei Kindobjekte Blätter (*Blatt*) sowie Komposita (*Kompositum*) sein können. Operationsaufrufe leitet die *Komponente* an das *Kompositum* oder *Blatt* weiter. Das *Kompositum* stellt den Behälter für die Kindobjekte dar und hält Referenzen auf diese. Es implementiert die Methoden zur Verwaltung der Kindobjekte (siehe Abbildung 10.4, *Kompositum*). Operationsaufrufe führt das *Kompositum* aus, indem es sie an alle seine Kindobjekte weiterleitet. Das *Blatt* implementiert letztlich diese Operation.

Konsequenzen

Das Kompositionsmuster definiert Klassenhierarchien auf Basis von Klassen für primitive und zusammengesetzte Objekte. Primitive Objekte können zu komplexeren Objekten zusammengesetzt werden, was wieder rekursiv fortgesetzt werden kann. Vor dem Klienten wird also verborgen, ob er mit einem primitiven Objekt oder einer Komposition von Objekten kommuniziert. Dadurch wird die Komplexität des Klienten vereinfacht, da er beim Zugriff keine Rücksicht auf strukturabhängige Besonderheiten nehmen muss.

Des Weiteren ist die Struktur leicht erweiterbar, da beim Hinzufügen neuer Blattklassen keine anderen Klassen angepasst werden müssen. Dies birgt jedoch den Nachteil, dass man keine Kontrolle darüber hat, welche Objekte in einem Kompositum zusammengefasst werden können. Es kann ja durchaus sein, dass man nicht möchte, dass bestimmte Objekte komponiert werden.

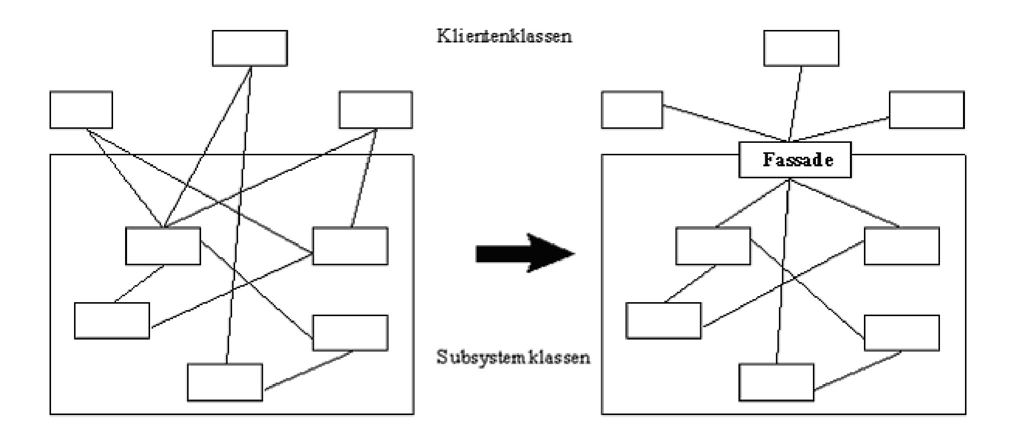


Abbildung 10.5.: Fassade

10.3.2 Fassade

Zweck

Es soll für Schnittstellen eines Subsystems eine einzige Schnittstelle bieten. Die Fassade deklariert eine abstrakte Schnittstelle und vereinfacht so die Benutzung des Subsystems.

Motivation

Ein System wird oft in Subsysteme unterteilt, um seine Komplexität zu reduzieren. Die Klientenklassen müssen jedoch alle Klassen des Subsystems kennen, auf deren Schnittstelle sie zugreifen wollen, um die Methoden einer bestimmten Klasse aufrufen zu können.

Das Fassademuster sieht ein Fassadeobjekt vor, welches eine einzelne vereinfachte Schnittstelle zu den Funktionen des Subsystems bietet.

Anwendbarkeit

Das Fassademuster findet Verwendung, wenn ein komplexes Subsystem von Klassen vorhanden ist, für das eine einfache Schnittstelle angeboten werden soll. Die Fassade bietet für den Klienten einen leichten Zugriff auf das Subsystem.

Wenn zwischen den Klienten und den Subsystemklassen viele Abhängigkeiten bestehen, sollte das Muster als Entkopplung des Systems eingesetzt werden, um die Unabhängigkeit zu fördern.

Ebenfalls eingesetzt werden sollte das Muster, wenn Subsysteme in Schichten aufgeteilt werden sollen. Die Fassade dient dann als Eintrittspunkt zu jeder Schicht. Dadurch werden die Abhängigkeiten zwischen den Schichten auf die Fassade begrenzt.

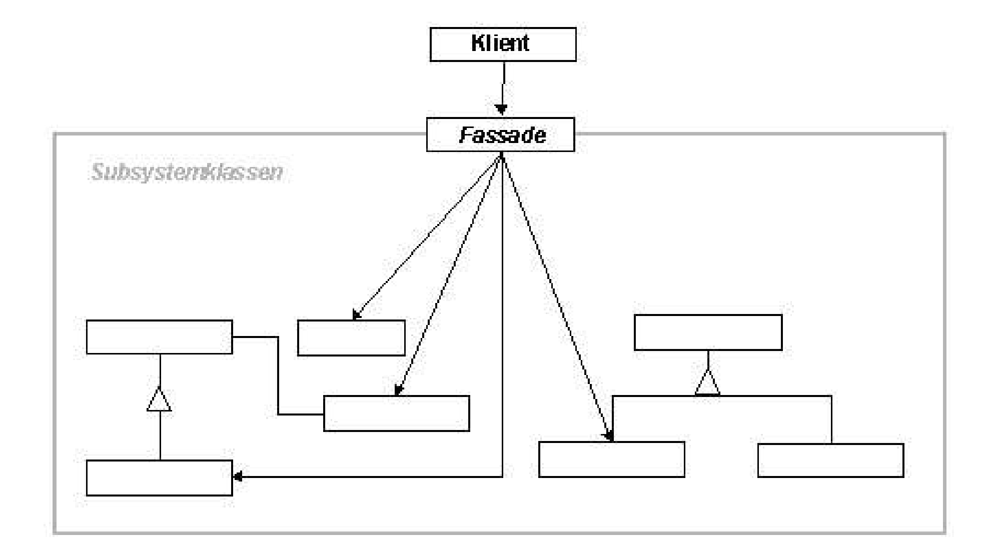


Abbildung 10.6.: Fassadenstruktur

Technische Realisierung

Der *Klient* nutzt die Schnittstelle der Klasse *Fassade*. Diese Klasse definiert die Schnittstelle für den Zugriff auf das Subsystem. Wenn der *Klient* eine Anfrage stellt, dann leitet die *Fassade* diese an das zuständige Subsystemobjekt weiter.

Die Subsystemklassen implementieren die Funktionalität des Subsystems und werden von der *Fassade* aufgerufen. Sie kennen die *Fassade* jedoch nicht, sie haben also keine Referenz auf die *Fassade*.

Konsequenzen

Das Subsystem wird durch die Reduzierung von zu verwaltenden Objekten einfacher nutzbar, da nur die Fassade verwaltet werden muss, die den Benutzer von den einzelnen Subsystemkomponenten abschirmt. Weiterhin wird die lose Kopplung zwischen Klienten und Subsystem gefördert, wodurch Komponenten des Subsystems leichter ausgetauscht werden können, so dass die Implementierung der Fassade angepasst werden muss.

Die Verwendung von Subsystemklassen wird jedoch nicht verhindert, so dass Anwendungen bei Bedarf weiterhin direkt auf das Subsystem zugreifen können. Dies lässt die Schlussfolgerung zu, dass sich durch das Fassademuster keine Nachteile ergeben.

10.3.3 Adapter

Zweck

Das Adaptermuster lässt Klassen zusammenarbeiten, die aufgrund inkompatibler Schnittstellen sonst nicht miteinander kommunizieren könnten. Das Muster dient also der Schnittstellenkonvertierung.

Motivation

Bei der Entwickelung einer Anwendung kann man sich eine Menge Arbeit ersparen, wenn man auf Funktionalitäten aus bereits bestehenden Toolkits zurückgreifen kann. Nehmen wir an, wir müssten einen Graphikeditor, den wir vor einiger Zeit für einen Arbeitgeber implementiert haben, um weitere Funktionen wie z.B. das Darstellen von Text in einer Graphik erweitern. Bisher war der Editor nur in der Lage mit einfachen Strukturen wie Kreisen oder Linien umzugehen, aber der Arbeitgeber meint, dass diese Erweiterung für uns kein Problem darstellen sollte, da es ja genügend Klassenbibliotheken gibt, die den Umgang mit Text erleichtern. Natürlich lässt sich so eine Klassenbibliothek auch finden, doch leider wird es wohl eher unwahrscheinlich sein, dass diese zu unserer bestehenden Schnittstelle für graphische Objekte passt. Wir könnten natürlich versuchen die Klassen so anzupassen, dass eine Kommunikation untereinander möglich ist, doch würde dies klar gegen den Gedanken der Wiederverwendbarkeit sprechen. Diese Möglichkeit der Abänderung besteht zumeist auch gar nicht, da bei einem bestehenden Toolkit es eher unwahrscheinlich ist, das wir Zugriff auf den Quellcode haben.

Eine Lösung für dieses Problem wäre, eine neue Klasse zu generieren, die wir als Adapter zwischen die inkompatiblen Klassen setzen können. In unserem Fall würde also diese Adapterklasse die Schnittstelle unserer Klasse für graphische Objekte implementieren, indem sie Methodenaufrufe an die neue Klassenbibliothek weiterleitet und daraus das gewünschte Verhalten definiert.

Diese gerade vorgestellte Lösungsidee spiegelt die Objektversion des Adaptermusters wieder. Nicht verschwiegen werden sollte, dass es auch noch eine zweite Variante gibt, die klassenbasiert arbeitet und Mehrfacherbung benutzt. Auf diese werden wir aber nicht weiter eingehen, da ihre Ideen und Ansätze nahezu identisch sind und wir uns vorwiegend mit der Programmiersprache „*Java*“ beschäftigen, die das Konzept der Mehrfacherbung nicht unterstützt.

Anwendbarkeit

Das Adaptermuster verwendet man bei inkompatiblen Schnittstellen von bereits bestehenden Klassen. Genau dies haben wir ja in unserem einleitenden Beispiel getan. Es kann auch angewandt werden, um bei der Erzeugung wieder verwendbarer Klassen zu helfen, da bei deren Entwurf nicht immer abschätzbar ist, wie diese später mit anderen Klassen zusammenarbeiten werden.

Technische Realisierung

Der *Klient* arbeitet mit Objekten, die sich an die umgebungsspezifische Schnittstellenstruktur der Klasse *Ziel* halten. *Ziel* definiert die vom Klienten benutzte Schnittstelle. Die *Adaptierte-Klasse* definiert eine zu der Klasse *Ziel* inkompatible Schnittstelle, die adaptiert werden muss, um sie für die Klasse *Ziel* nutzbar zu machen. Diese Anpassung findet nun über den *Adap-*

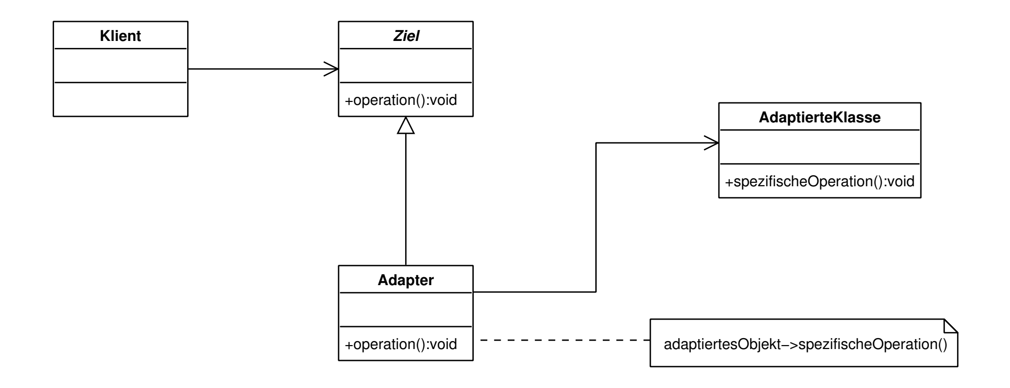


Abbildung 10.7.: Adapterstruktur

ter statt, welcher die Methoden der Klasse *AdaptierteKlasse* benutzt, um das vom Klienten erwartete Verhalten zu erfüllen.

Der *Klient* nutzt also Adapaterobjekte, um seine Anfragen bearbeiten zu lassen.

Konsequenzen

Ein Objektadapter kann mit mehreren adaptierten Klassen zusammenarbeiten, nämlich mit der adaptierten Klasse selbst und all ihren Unterklassen. Er kann weiterhin neue Funktionalität zu allen adaptierten Klassen hinzufügen. Diese neue Funktionalität wird im Adapter implementiert. Erschwert wird allerdings die Überschreibbarkeit von Verhalten der adaptierten Klasse, da im Objektadapter folgender Umweg gegangen werden muss:

Um das Verhalten der zu adaptierenden Klasse zu überschreiben fügt man eine neue Klasse hinzu, die von der zu adaptierenden Klasse erbt und das Verhalten überschreibt. Der Adapter muss anschließend die neue Klasse adaptieren.

Der Implementierungsaufwand kann durchaus unterschiedlich ausfallen. Es richtet sich danach, wie verschieden die anzupassenden Schnittstellen zueinander sind. Der Aufwand kann von einfachem Anpassen der Methodennamen bis zu komplexen Algorithmen zur Umwandlung reichen.

Ferner kann es problematisch sein, dass der Adapter nicht für alle Klienten transparent ist. Ein durch Adapter angepasstes Objekt ist nicht mehr konform zu der Schnittstelle der adaptierten Klasse. Aus diesem Grund kann ein adaptiertes Objekt nicht ohne weiteres wie ein nicht adaptiertes Objekt der adaptierten Klasse eingesetzt werden.

10.3.4 Vergleich zwischen Fassade und Adapter

Das Fassadenentwurfsmuster kann als Adapter für eine größere Anzahl von Klassen angesehen werden. Dies steht im Gegensatz zu dem „ein Klassenadapter“ der durch das Adaptermuster realisiert wird. Der Begriff „ein Klassenadapter“ soll hierbei bedeuten, dass wir uns nur mit der Adaptierung einer Klasse beschäftigen und nicht eine einheitliche Schnittstelle für

ein Subsystem realisieren wollen. Dieser Vergleichsansatz verbirgt jedoch, dass das Adaptermuster zwei bereits existierende Schnittstellen kompatibel zueinander macht, wobei hingegen das Fassadenmuster eine neue Schnittstelle für das Subsystem einführt.

10.3.5 Zusammenfassung

Wir haben an dieser Stelle natürlich nur einen kleinen Teil der Strukturmuster vorgestellt. Beim Studium eines Buches über Entwurfsmuster, wie das Werk der „Gang of Four“ (Gamma u. a., 2001) oder auch das Buch von Buschman (Buschmann u. a., 1996), wird man feststellen, dass es natürlich noch einige Strukturmuster mehr gibt. Unsere Auswahl soll deshalb nur einen repräsentativen Überblick darüber geben, mit welchen Konzepten diese Muster arbeiten, um ihre Ziele zu erreichen. Unser Anliegen, war vielmehr aufzuzeigen, dass sich die Muster, obwohl sie auf einer kleinen Menge von Entwurfsmechanismen beruhen, stark in ihrer Zielsetzung unterscheiden. Auch wenn die Muster in ihrer Struktur sehr ähnlich sind, verfolgen sie doch immer unterschiedlich geartete Ziele. Wichtig ist halt nur, dass man bei der Benutzung von Strukturmustern besonders darauf achten sollte, welche Zielsetzung für uns relevant ist.

10.4 Verhaltensmuster

Bei der Software-Entwicklung von objektorientierter Systeme müssen Entwickler die wichtige Entscheidung treffen wie die Verantwortung zwischen Objekten verteilt werden soll.

Man kann sich im Grunde für zwei Arten der Verteilung entscheiden, wobei beide ihre Nachteile haben. Zum einen kann man sich für viele einfache Objekte entscheiden. Doch dies würde zu einer Vielzahl von Objektinteraktionen führen, was nicht immer leicht durchschaubar und schwer wartbar ist. Zum anderen kann man sich aber auch für wenige, dafür aber mit umfangreichen Funktionen ausgestattete Objekte entschließen. Dieses Vorgehen hat dann zwar weniger Interaktion untereinander zur Folge, kann aber trotzdem schwer wartbar sein, da die Objekte schnell monolithisch werden. Darüber hinaus würde keine der Möglichkeiten zu einem wiederverwendbaren System führen. Eine Vielzahl von Abhängigkeiten zwischen Objekten und monolithische Objekte machen die Wiederverwendung einzelner Objekte schwierig.

Verhaltensmuster helfen, diese Probleme zu lösen. Das bereits kurz im Zuge von MVC angesprochene Beobachtermuster kapselt die Interaktion zwischen Objekten und fördert damit die lose Kopplung zwischen ihnen. Andere Verhaltensmuster, wir werden gleich zwei von ihnen genauer kennen lernen, helfen dabei Verhalten zwischen Objekten zu verteilen, wodurch eine Wiederverwendung der Objekte wahrscheinlicher wird.

Wie schon bei den Strukturmustern werden wir uns auch an dieser Stelle nur mit den objektbasierten Verhaltensmustern befassen.

10.4.1 Besucher

Zweck

Eine auf den Elementen einer Objektstruktur auszuführende Operation wird als eigenständiges Objekt gekapselt. So ermöglicht das Muster die Einführung einer neuen Operation, ohne die Klassen der von ihr bearbeiteten Elemente zu verändern.

Motivation

Eine Objektstruktur soll um Operationen erweitert werden. Dabei müssten alle Objekte, um die entsprechende Operation erweitert werden, sofern sie auf allen Objekten benötigt wird. Um zu verhindern, dass alle Objekte geändert werden müssen, kapselt das Besuchermuster die Operationen jeder Klasse als separates Objekt. Dieses Objekt wird den Elementen bei Bedarf übergeben.

Anwendbarkeit

Das Besuchermuster wird verwendet bei Klassen einer Daten- oder Objektstruktur, die sich nicht oder nur sehr selten ändern, die auf ihnen auszuführenden Operationen sich hingegen häufig ändern.

Ebenso benutzt man das Besuchermuster, wenn man eine große Anzahl von Operationen hat, die auf den Objekten eine Objektstruktur ausgeführt werden müssen. Die verwandten Operationen können dann aus den Objekten herausgenommen werden und in eigene Objekte gekapselt werden.

Technische Realisierung

Die abstrakte Klasse *Besucher* definiert die Schnittstelle für alle konkreten Besucher (z.B. *KonkreterBesucher1*). Die Namen der Operationen der Schnittstelle sowie deren Signatur benennen die Klasse, welche die Besuche-Operation des Besuchers aufruft. Der konkrete Besucher implementiert die vom Besucher zuvor definierten Operationen. Will ein Klient eine bestimmte Operation auf der Objektstruktur ausführen, dann erzeugt er einen konkreten Besucher und übergibt ihn an die Objektstruktur.

Die Objektstruktur hält die Elemente, auf die die Besucher angewendet werden können. Bei der Anwendung eines Besuchers, werden alle Elemente der Objektstruktur von diesem besucht. Die abstrakte Klasse *Element* definiert die Schnittstelle für die konkreten Elemente. Diese Schnittstelle spezifiziert die Operation „*nimmEntgegen*“ die von den konkreten Elementen implementiert wird. Über diese Operation nimmt ein konkretes Element einen an die Objektstruktur gerichteten Besucher entgegen. Darauf hin ruft das Element die ihm entsprechende Operation des Besuchers auf und übergibt dabei eine Referenz auf sich selbst. Des Weiteren implementieren die konkreten Elemente Operationen für den Zugriff auf ihre internen Daten, damit die konkreten Besucher ihre Aufgaben auf den konkreten Elementen durchführen können. So erhält der Besucher Zugriff auf das entsprechende Element, ohne das dabei die Kapselung des Elementes verletzt wird.

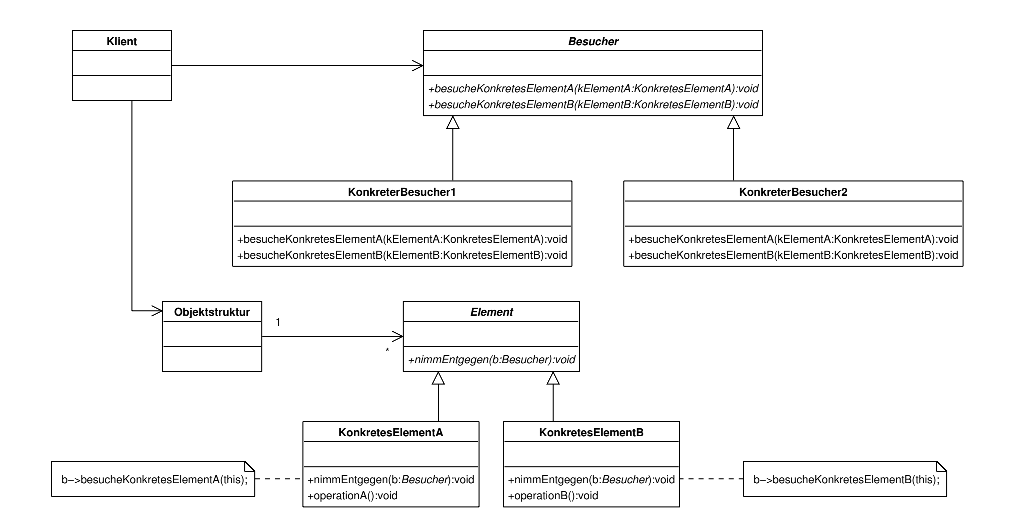


Abbildung 10.8.: Besucherstruktur

Konsequenzen

Das Besuchermuster erlaubt es, neue Operationen auf einfache Weise zu einer Objektstruktur hinzuzufügen, indem einfach ein neuer Besucher eingeführt wird. Die Klassen der Objektstruktur müssen dabei nicht angepasst werden. Ebenso verhindert das Muster, dass Operationen, die ein ähnliches und zusammengehörendes Verhalten implementieren, über die Klassen der gesamten Objektstruktur verteilt werden. Diese zusammengehörenden Operationen werden dazu in einer Besucherklasse gekapselt. Dieses Ausgliedern der Operationen aus den Klassen der Objektstruktur macht es unnötig, alle diese Klassen neu zu übersetzen, wenn sich an einer gemeinsamen Operation etwas ändert.

Es ist hingegen aufwendig, Klassen zu der Objektstruktur hinzuzufügen, da eine neue Klasse in der Objektstruktur eine neue Operation und deren Implementierung in jedem Besucher verlangt. Daher sollte man sich vor Gebrauch des Besuchermusters Gedanken darüber machen, ob sich die Objektstruktur voraussichtlich ändern wird oder nicht.

Da die Methoden der Besucher die Veränderungen an den Objekten der besuchten Struktur vornehmen, ist es natürlich nötig, dass die Eigenschaften der Objekte von außen zugänglich sind. Dies ist ein Nachteil des Besuchermusters, denn hierdurch wird die Kapselung der Objektstrukturdaten aufgebrochen.

10.4.2 Iterator

Zweck

Das Iterator-Pattern zeigt eine Möglichkeit auf, auf zusammengesetzte Objekte sequentiell zuzugreifen, ohne dabei deren innere Struktur preiszugeben.

Motivation

Der sequentielle Zugriff auf zusammengesetzte Objekte bzw. Aggregate von Objekten erfordert exaktes Wissen über Implemetierungsdetails des komplexen Objekts. Bei solch einer zusammengesetzten Struktur (z.B. Liste) ist es wünschenswert, auf die Elemente zugreifen zu können, ohne dabei „*Interna*“ offen legen zu müssen. Außerdem sollte die Möglichkeit bestehen, die Struktur auf verschiedene Arten und zeitgleich mehrfach zu traversieren.

Um dies zu realisieren, bedient sich das Iteratormuster eines Iterator-Objekts, welches die Funktionalität zur Traversierung zugewiesen bekommt, so dass diese Funktionalität aus der Struktur des zusammengesetzten Objekts entfernt werden kann. Dies bietet den Vorteil, dass die Schnittstelle des traversierten Objekts kompakt gehalten wird, auch wenn verschiedene Traversierungsarten definiert werden.

Anwendbarkeit

Das Iteratormuster wird verwendet um den Zugriff auf zusammengesetzte Objekte zu ermöglichen ohne dabei „Interna“ preiszugeben. Mit diesem Muster lassen sich dann auch Möglichkeiten bieten auf demselben zusammengesetzten Objekt verschiedene Arten der Traversierung anzubieten. Ein Beispiel hierfür wäre es auf einer Baumstruktur „in-order“ oder „post-order“ Durchlauf-Mechanismen anzubieten, je nachdem was benötigt wird. Es bietet sich des Weiteren an dieses Muster zu verwenden, wenn man eine einheitliche kompakte Schnittstelle auf die Traversierungsmethoden unterschiedlicher zusammengesetzter Objekte anbieten will. Also bei so verschiedenen Strukturen wie Listen oder Bäumen, trotzdem eine gemeinsame Schnittstelle für den Zugriff haben möchte.

Technische Realisierung

Die Klasse *Iterator* definiert eine Schnittstelle für den Zugriff und zur Traversierung von Elementen. Der *KonkreteIterator* implementiert diese Schnittstelle. Er verwaltet die aktuelle Position während der Traversierung des Aggregats. Die Klasse *Aggregat* definiert eine Schnittstelle für die Erzeugung eines Objekts der Klasse Iterator. Wir lassen an dieser Stelle bewusst aus, dass in der Schnittstelle von *Aggregat* natürlich auch noch der verwaltende Teil, also wie Elemente zu einem Aggregat hinzugefügt oder entfernt werden, definiert wird. Das *KonkreteAggregat* implementiert die Operationen zum Erzeugen eines konkreten Iterators, indem es ein Objekt der passenden KonkreterIterator-Klasse an den Klienten zurückgibt. Der Klient verwendet nun ausschließlich die Schnittstellen-Operationen des Iterators um seine gewünschten Traversierungsfunktionalitäten zu erhalten, ohne dabei irgendwelche Implementierungsdetails zu benötigen. Dieser *KonkreteIterator* verwaltet nun die aktuelle Position im *Aggregat* und kann die nachfolgende Position errechnen.

Bevor wir nun zu den Konsequenzen des Iteratormusters kommen, wollen wir noch erwähnen, dass durchaus einige Variationen dieses Musters existieren. Diese Variationen beschäftigen sich z.B. mit der Frage nach der Robustheit eines Iterators, also wie er auf Änderungen der Struktur des Aggregats während der Traversierung reagiert. Dies könnte den Iterator ja durchaus verwirren. Solch ein robuster Iterator garantiert, dass eine Traversierung von eventuellen Einfüge- und Löschoperationen im Aggregat nicht gestört wird, ohne das Aggregat vorher zu kopieren.

Eine andere Möglichkeit der Variation, die man beim Iteratormuster hat, besteht darin sich

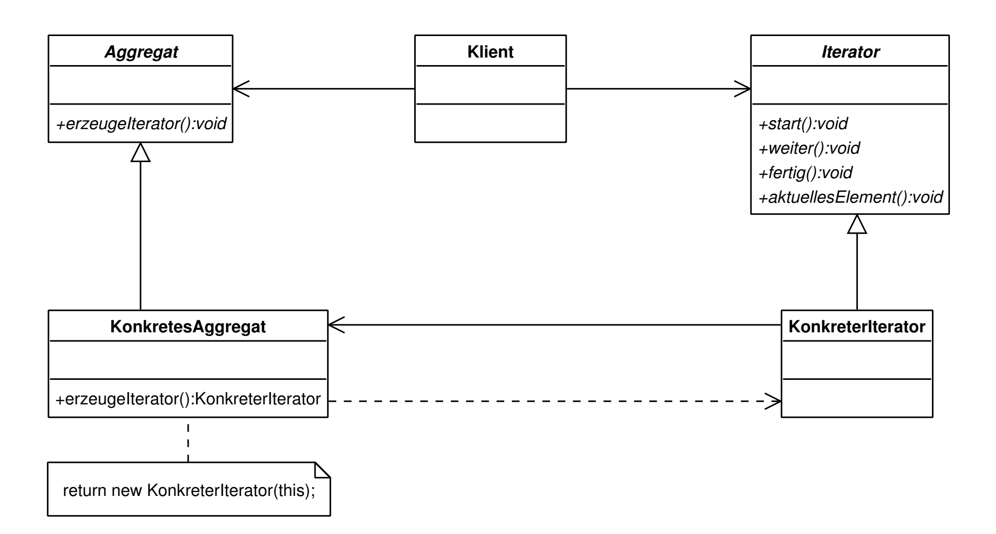


Abbildung 10.9.: Iteratorstruktur

für interne oder externe Iteratoren zu entscheiden. Der Namensgebung intern oder extern wird dadurch geprägt wer die Kontrolle über die Traversierung hat. Wenn der Client die Kontrolle hat, spricht man von einem externen Iterator (d.h. die Steuerung ist extern zum Iterator). Wenn dagegen der Iterator selbst die Kontrolle behält, spricht man von einem internen Iterator. Beim Verwenden eines externen Iterators, muss der Klient das jeweils aktuelle Element des Aggregates explizit anfordern und den Schritt zur nächsten Position explizit verlangen. Bei internen Itertoren dagegen überreicht der Client dem Iterator eine Operation, die an jedem Element auszuführen ist und lässt ihn machen; der Iterator ist in diesem Fall sehr selbständig. Auf weitere bekannten Variation dieses Entwurfsmusters wollen wir nicht eingehen, sondern nur auf das Buch der „Gang of Four“ (Gamma u. a., 2001) verweisen.

Konsequenzen

Die Traversierung eines Aggregates mit Iteratoren macht es unnötig, Traversierungsmethoden in der Schnittstelle von Aggregate vorzusehen. Dadurch wird die Komplexität der Aggregatschnittstelle vereinfacht. Da jeder Iterator die Information über seine aktuelle Position in seinen Zustandsdaten behält, kann mehr als ein Iterator auf einem Aggregat aktiv sein. Hätte hingegen die Aggregateschnittstelle Traversierungsmethoden, müssten diese Methoden ihre aktuelle Position in den Zustandsdaten des Aggregats speichern. Daraus würde sich allerdings ergeben, dass zwei simultane Traversierungen sich gegenseitig stören. Ein weiterer Vorteil der sich beim Verwenden des Iteratormusters ergibt ist, dass die Art der Traversierung leicht geändert werden kann, indem das Iteratorobjekt durch ein anderes ersetzt wird.

10.4.3 Vergleich zwischen Besucher und Iterator

Beim Iterator wird über die Objekte in einem Aggregat traversiert, wohingegen beim Besucher über die gesamte Objektstruktur traversiert wird.

10.4.4 Zusammenfassung

Wie schon bei den Strukturmustern ist die von uns getroffene Auswahl sehr klein im Gegensatz zu den bisher verfügbaren Verhaltensmustern. Aber die Auswahl der Muster braucht auch nicht so umfangreich zu sein, um zu erkennen, dass Verhaltensmuster im Allgemeinen gut miteinander kombinierbar sind. Sie schließen sich also nicht gegeneinander aus, sondern unterstützen sich oftmals gegenseitig. Die beiden von uns in diesem Abschnitt besprochenen Muster (Besucher und Iterator) könnten z.B. so miteinander arbeiten, dass Iterator dazu eingesetzt wird um über eine zusammengesetzte Struktur zu traversieren, wobei das Besuchermuster dann Operationen für jedes besuchte Element bereithält.

Der Zusammenarbeitsaspekt erstreckt sich aber natürlich nicht nur auf die Klasse der Verhaltensmuster untereinander; z.B. lassen sie sich auch mit Strukturmustern sehr schön kombinieren. Ein Klient, der das Kompositionsmuster dazu verwendet sein Objektstruktur zu verwalten, könnte dann das Besuchermuster dazu verwenden, um Operationen auf seiner Objektstruktur zu implementieren.

10.5 Schlussfolgerungen

Nach dieser kurzen Einführung in die Welt der Entwurfsmuster wollen wir nun Zusammenfassen, warum es Sinn macht, sich mit ihnen zu beschäftigen. Bei dem Entwurf von objektorientierter Software kann man auf eine Vielzahl von Problemen stoßen, die gelöst werden müssen. Welche Erkenntnisse, die uns bei dem Ziel unterstützen diese Probleme zu lösen, haben wir nun dazu gewonnen?

Objektorientierte Software besteht aus Objekten, den Instanzen von Klassen. In einem Objekt werden Daten und Operationen zusammengefasst. Die Operationen der Objekte werden auf Anfrage eines Klienten ausgeführt und arbeiteten auf den Daten des Objekts. Die Anfragen des Klienten stellen die einzige Möglichkeit dar, ein Objekt dazu zu bewegen, eine seiner Operationen auszuführen. Die Daten des Objekts lassen sich ihrerseits nur über die Operationen des Objekts verändern. Dies ist der Grund dafür, dass der Zustand des Objekts von außen nicht sichtbar ist, was man allgemein als Kapselung der internen Daten versteht. Ein nicht triviales Problem bei der Softwareentwicklung ist es nun, herauszufinden welche Aufgaben auf mehrere Objekte verteilt und welche in einzelnen Objekten gekapselt werden sollten. Natürlich kann es aber auch sein, dass man noch damit zu kämpfen hat überhaupt erst einmal eine geeignete Objektrepräsentation seiner Vorstellungen zu finden. Dies ist oftmals gar nicht so einfach, da in der objektorientierten Software sehr oft Klassen vorkommen die in der realen Welt keine Entsprechungen haben, die man also nicht intuitiv anlegen würde.

Genau hierbei können uns Entwurfsmuster helfen, die nicht immer offensichtlichen Abstraktionen und die dazu passenden Objekte zu finden. Das unter den Strukturmustern vorgestellte Kompositum soll an dieser stelle als Beispiel dienen. In diesem Muster haben wir

einen Weg gefunden, wie wir einfache Objekte, sowie Kompositionen von ihnen, auf genau die gleiche Art verwalten und benutzten können. Wir haben also erreicht, dass wir nicht zwischen unterschiedlichen Dingen unterscheiden müssen, die in der realen Welt durchaus einer Unterscheidung bedürfen.

Ein weiterer Ansatz, bessere objektorientierte Software zu entwickeln besteht darin, sich mit der Granularität von Objekten zu befassen. Mit dem Begriff Granularität sei hier gemeint, wie „Groß und Mächtig“ ein Objekt sein soll, d.h. wie viele Aufgaben es übernehmen kann. In einem Softwareentwurf können Objekte der verschiedensten Granularitäten vorkommen, vom „Strohhalm“ bis zum „Heuhafen“ kann man sich alles vorstellen. Wie findet man nun eine ideale Granularität für sein Projekt?

Es werden also Kriterien benötigt, die uns bei der Entscheidung helfen, was zu einem Objekt gemacht werden soll. Zu unserer Erleichterung gibt es Strukturmuster, die sich genau mit der Granularität von Objekten befassen. Wir haben bereits ein Muster kennen gelernt, dass in diese Kategorie passt: das Fassadenmusster. Es ist für Objekte zuständig, die ganze Subsysteme repräsentieren wollen (also einen geordneten und leichte Zugriff auf den „Heuhaufen“ bieten).

Ein Schlüssel zur erfolgreichen Wiederverwendung von Software liegt in der Erkenntnis, dass sich die Anforderungen an eine Anwendung oder an eine Klassenbibliothek mit der Zeit wandeln können. Mann ist damit gut beraten, diesen Wandel von Anfang an mit einzuplanen, damit ihre Systeme diese Evolution mitmachen können. Software sollte also in der Lage sein sich neuen Gegebenheiten anzupassen, ohne dabei zeit- und kostenintensives Neudesign nötig zu machen. Entwurfsmuster stellen sicher, dass solche Änderungen möglich bleiben, da sie die Veränderbarkeit von bestimmten Teilen des Systems erlauben, ohne auf andere Teile dabei Einfluss zu nehmen. Es wird also eine gewisse Robustheit für Änderungen geschaffen.

Bereits mit den wenigen Mustern, die wir nun kennen gelernt haben, ist schon eine grosse Fülle von Änderungsvorhersehbarkeiten möglich. Mit dem kurz angerissenen Muster *Abstrakte Fabrik* lassen sich z.B. Objekte indirekt erzeugen, wodurch wir die Möglichkeit erhalten, die Implementation später zu verändern, ohne dabei die Schnittstelle anpassen zu müssen. Des Weiteren können wir durch dieses Muster Hard- und Softwareplattformabhängigkeiten begrenzt werden. Ein anderes von uns diskutiertes Muster (Iterator) erlaubt es, dass algorithmische Abhängigkeiten vermieden werden können. Die Erweiterung der Flexibilität durch Komposition statt Vererbung erreichen wir mit dem Kompositummuster. Wie man sieht, haben wir nun schon ein stattliches Rüstzeug für Änderungen erhalten.

Zu guter letzt wollen wir aber nicht verschweigen, dass Entwurfsmuster kein „Allheilmittel“ sind. Es ist nur sinnvoll sie anzuwenden, wenn man sich im Klaren darüber ist, welche Vor- und Nachteile mit ihnen verbunden sind. Entwurfsmuster bringen i.d.R. Flexibilität und sollten auch nur dann eingesetzt werden, wenn diese Flexibilität auch gebraucht wird. Der Trade-off ist fast immer, dass die Struktur unseres Entwurfs komplizierter wird und man meist auch mit Performanzeinbußen leben muss. Entwurfsmuster sind also keine fertige Schablone für ein zu erstellendes Programm.

KAPITEL 11

Java3D

André Kupetz, Jan Wessling

11.1 Einleitung

Das Referat soll einen Überblick über die Programmierung in Java3D geben. Dabei wird insbesondere auf die Hintergründe eingegangen, die allgemein einer Programmierung von 3D-Szenen zu Grunde liegen.

Zu diesem Zweck werden hier als erstes die mathematischen Grundlagen kurz angesprochen. Des weiteren werden die Grundlagen diverser visueller Verfahrenstechniken und deren Verwendung vorgestellt und erläutert. Die darauf folgenden Kapitel beschäftigen sich mit dem Aufbau von Java3D, dessen Klassenbibliotheken, dem zentralen Begriff des Scenegraphen und verschiedenen Konstrukten in ihm. Abschließend wird auf die Realisierung einiger einfacher Beispiele eingegangen.

11.2 Grundlagen

Bevor wir mit Java3D beginnen, müssen wir erst ein paar mathematische Grundlagen des dreidimensionalen Raumes ins Gedächtnis zurückrufen. Dabei setzten wir als bekannt voraus, was Vektoren, Matrizen, Linien und Ebenen sind. Danach werden hier ein paar Renderingtechniken erklärt, die Java3D verwendet.

11.2.1 Mathematische Grundlagen

Wir werden jetzt als erstes einige der notwendigen mathematischen Grundlagen, die für die 3D-Visualisierung benötigt und in Java3D verwendet werden, behandeln. Dies sind insbesondere Polygone, homogene Koordinatensysteme und verschiedene Transformationen, wie Translation, Rotation, Skalierung, Spiegelung und Projektion.

Polygone

Polygone sind Vielecke, die allerdings nicht regelmäßig ausgerichtet sein müssen. Sie können als Tupel von Vektoren (d. h. als Matrix ihrer Einzelpunkte) dargestellt werden. Dabei müssen alle Punkte in derselben Ebene liegen (bei Dreiecken automatisch erfüllt).

Homogene Koordinatensysteme

Homogene Koordinatensysteme haben, ohne irgendwelche geometrische Bewandnis (Schiedermeier, 2002), den Zweck, geometrische Transformationen von Punkten (Translation, Rotation, Skalierung, Spiegelung, Projektion) allein durch Matrizen auszudrücken, und damit einheitlich zu behandeln. Weiterhin ist es damit auch möglich, mehrere Transformationen durch Matrizenmultiplikation zu verknüpfen. Dabei ist zu beachten, daß die Matrizenmuliplikation zwar assoziativ aber nicht kommutativ ist. Um eine Transformation auszuführen, muß lediglich die Transformationsmatrix mit dem Punkt multiplizert werden, um so den transformierten Punkt zu bekommen: $p' = p * A$.

Dem dreidimensionalen kartesischen Vekorraum mit x-, y- und z-Koordinate entspricht dabei ein homogener (x,y,z,w)-Vektorraum. Dabei entspricht einem homogenen Vektor (x,y,z,w) ein kartesischer Vektor $(x/w, y/w, z/w)$, falls $w \neq 0$. Damit ist auch klar, das Vielfache des Vektors (x, y, z, w) wieder den selben Punkt beschreiben. In der Computergrafik wird üblicherweise $w = 1$ für Punkte angenommen. Diese Punkte nennt man homogenisiert. Für Vektoren ist $w = 0$. Der Vorteil hier ist, das sich Translationen nur auf Punkte und nicht auf Vektoren auswirken, da man einen Vektor, also eine Richtung, nicht verschieben kann, während sich Rotation und Skalierung sehr wohl auch auf Vektoren auswirken.

Die unterste Zeile der von Transformationsmatrizen $(0, 0, 0, 1)$ ändert sich nur bei perspektivischen Transformationen. Sie enthält an den ersten drei Stellen einen Perspektivtransformationsanteil und an der letzten Stelle einen Skalierungsfaktor. Nur durch diese letzte Zeile wird später eine perspektivische Transformation ausgedrückt. Dabei entsteht meist ein nicht homogenisierter Punkt (x, y, z, w) mit $w \neq 1$. Um ihn zu homogenisieren, ist jedes Element durch w zu teilen.

Translation

Unter einer Translation versteht man das Verschieben eines Punktes/Objektes um Δx, Δy, Δz. Diese Transformation läßt sich durch eine Vektoraddition in Form von $(a, b, c) + (\Delta a, \Delta b, \Delta c) = (a + \Delta a, b + \Delta b, c + \Delta c)$ ausdrücken oder durch die homogene Translationsmatrix. Sie ist

$$A = \begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Skalierung und Spiegelung

Eine Skalierung bedeutet eine Streckung oder Stauchung der Koordinaten eines Punktes/Objektes bzgl. des Koordinatenursprungs. (Brüderlein und Meier, 2000, S. 41) Die zugehörige homogene Matrix ist

$$A = \begin{bmatrix} F_x & 0 & 0 & 0 \\ 0 & F_y & 0 & 0 \\ 0 & 0 & F_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

F_x, F_y, F_z sind dabei die Skalierungsfaktoren, mit denen in die jeweilige Richtung gestreckt wird. Durch negative Skalierungsfaktoren drückt man eine Spiegelung aus (z. B. ist $F_x = F_y = -1$ eine Spiegelung am Ursprung).

Rotation

Beim Rotieren im dreidimensionalen muss man sich für eine Achse entscheiden, um die man das Objekt rotieren will. So ist eine Rotation um die z-Achse mit dem Winkel θ durch

$$A_{z\theta} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

gegeben. Analog dazu existieren auch für die anderen Achsen

$$A_{x\theta} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

und

$$A_{y\theta} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Eine Rotation um beliebige Achsen kann man durch Translationen und Rotationen darstellen. Dazu bringt man erst die Drehachse per Translation und Rotation auf eine Koordinatenachse, dreht um den gewünschten Winkel, und führt die inversen Funktionen der Vorbereitungsschritte aus.

Projektionen

Projektionen gehören von der Art ihrer Matrix zu den Skalierungen. Dabei wird das 3-dimensionale Bild in eine Ebene projeziert. Dies geschieht mit der Fluchtpunktperspektive (man erinnere sich an den Kunstunterricht) und heißt demnach auch perspektivische Projektion. Ein Spezialfall ist die rechtwinklige Projektion. Sie bewirkt die „Nullsetzung“ einer Koordinate und damit eine Projektion aus dem n-dimensionalen in den n-1 dimensionalen Raum. Die Matrix hätte bei dieser Projektion in die x-y Ebene im dreidimensionalen dann den Wert $F_z = 0$ (siehe Skalierungsmatrix).

11.2.2 Farben, Beleuchtung und Renderingtechniken

Um Renderingtechniken vorstellen zu können, muß man wissen, wie die Farben dargestellt werden, und wie sich Beleuchtung auf Objekte auswirkt. Dies wird gleich als erstes vorgestellt. Danach wird auf verschiedene Schattierungs- und Textur-Mapping-Verfahren eingegangen. Die Schattierungsverfahren berechnen, wie ein Objekt aufgrund seiner Position im Raum, seiner Farbe und der vorhandenen Beleuchtung (Licht und Schatten) gezeichnet wird, während es bei Textur-Mapping darum geht, eine Textur auf ein 3D-Objekt zu legen.

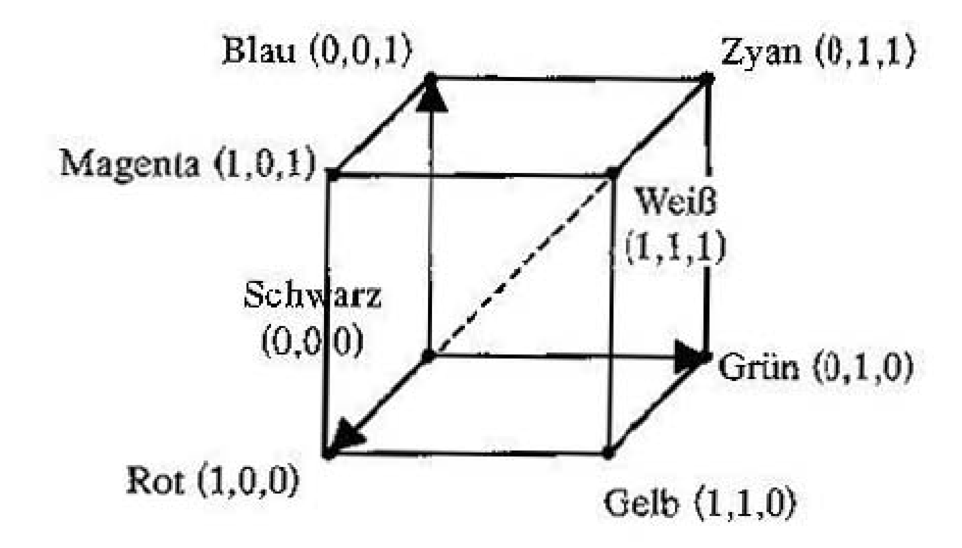


Abbildung 11.1.: Der RBG-Würfel (Brüderlein und Meier, 2000)

Das RGB Farbmodell

Java3D stützt sich auf das RGB-Modell. Hierbei wird jede Lichtquelle in drei Farbanteile zerlegt. Es entstehen die drei Lichtquellen der Farben Rot, Grün und Blau. Sie können als unabhängige Koordinaten $< R, G, B >$ eines dreidimensionalen Raumes wie in Abb. 11.1 dargestellt werden. Die Werte der Koordinaten zwischen 0 und 1 entsprechen der relativen Lichtstärke der einzelnen Quellen R, G, B zwischen 0% und 100%. Der Koordinatenursprung entspricht dabei schwarz, der Punkt $< 1, 1, 1 >$ entspricht weiß. Weitere Farbmodelle sind das CMY-Modell und das HSV-Modell, allerdings werden beide in Java3D nicht benutzt und daher hier auch nicht erklärt.

Beleuchtungsmodell

Das in Java3D benutzte Beleuchtungsmodell ist das Beleutungsmodell nach Phong. Es unterscheidet zwischen diffuser, spekulativer und ambienter Reflexion. Dabei wird die Leuchtdichte I, das Maß für die Helligkeit einer beleuchteten Oberfläche bzw. einer Lichtquelle, an einem Punkt berechnet.

Diffuse Reflexion ergibt sich, wenn das Licht nicht direkt von der Lichtquelle reflektiert wird, sondern erst etwas unter die Oberfläche eindringt und von dort in alle Richtungen reflektiert wird. Dabei ist der Blickwinkel des Betrachters unerheblich. Es kommt alleine auf den Einfallswinkel der Lichtquelle an.

Bei spekulativer Reflexion wird angenommen, daß das Licht von der Oberfläche wie von einem nicht idealen Spiegel gestreut reflektiert wird. Matte Oberflächen streuen dabei mehr als glänzende Oberflächen, wie es in Abb. 11.2 dargestellt ist. Bei der ambienten Reflexion besitzt das Licht keine einheitliche Richtung, sondern es wird aus allen Richtungen in alle Richtun-

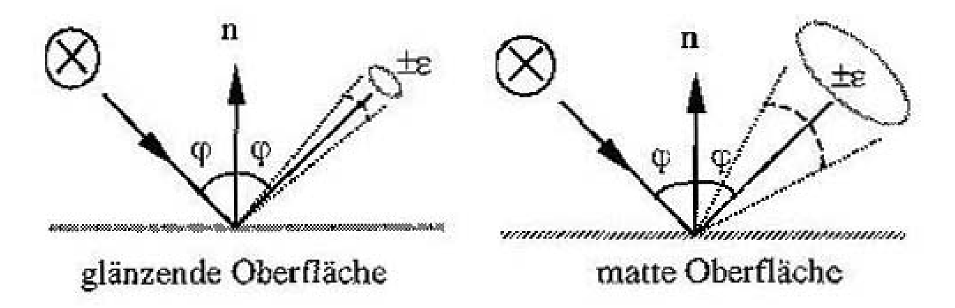


Abbildung 11.2.: Glänzende und matte Reflexion (Brüderlein und Meier, 2000)

gen gestrahlt. Dies entspricht annähernd der Beleuchtungssituation bei bedecktem Himmel. Ambientes Licht ist in den meisten Umgebungen mehr oder weniger vorhanden, weil auch gerichtetes Licht an Partikeln der Luft und an Gegenständen in alle Richtungen gestreut wird. Das vollständige Phong-Modell ergibt sich damit zu:

$$I = I_0 * (R_{amb} + R_{diff} * \cos\varphi + R_{spec}(\varphi) * \cos^n\theta)$$

φ ist dabei der Einfallswinkel der Lichtquelle und θ ist die Abweichung des Betrachters vom idealen Ausfallwinkel. Die drei Reflexionskoeffizienten für die unterschiedlichen Reflexionsarten ($R_{amb}, R_{diff}, R_{spec}$) und der Exponent n sind die Parameter, die das Aussehen eines Objektes bestimmen. So ist z. B. bei n = 1 eine sehr breite Streuung gegeben, während man, um glänzende Oberflächen und damit eine schmale Streuung darstellen will, ein großes n braucht.

Es muß klar gestellt werden, daß das Phong-Modell nur eine Annäherung an die Realität sein kann. So sind insbesondere metallene Oberflächen in ihrem Reflexionsverhalten viel komplexer.

Painter's Algorithmus

Ein sehr einfaches, wenngleich beschränktes Schattierungsverfahren ist der sogenannte Painter's Algorithmus (Brüderlein und Meier, 2000, S. 129). Es wird dabei eine Maltechnik für Deckfarben algorithmisch nachvollzogen. Bei dieser Technik werden zuerst die weiter hinten gelegenen Objekte gemalt. Nach und nach werden diese durch weiter vorne liegende Objekte ganz oder teilweise verdeckt.

Dieses Verfahren funktioniert in vielen Fällen ganz gut, jedoch können in manchen Szenen Probleme auftreten, da die Objekte im dreidimensionalen Raum nicht in eine eindeutige räumliche Reihenfolge gebracht werden können und damit sich überschneidende Objekte nicht richtig gezeichnet werden können.

Z-Buffer-Verfahren

Beim Z-Buffer-Verfahren wird zu jedem Punkt eines Polygons, und nicht wie im Painter's Algorithmus zu jedem Objekt, individuell eine Tiefeninformation (z-Wert) berechnet und zusätzlich zur Farbinformation abgespeichert. Die Sichtbarkeitsbestimmung kann dann pixel-

weise geschehen. Dazu wird erst die Farbe aller Pixel auf die Hintergrundfarbe gesetzt und der z-Wert aller Pixel wird auf das Maximum gesetzt. Dann werden für alle Polygone während der Rasterumwandlung die z-Werte mitberechnet. Beim Abspeichern in den Bildspeicher wird dann jedes Pixel mit dem im Bildspeicher aktuellen Pixel verglichen. Der Pixel mit dem niedrigeren z-Wert, also der Pixel der näher am Betrachter ist, wird in den Bildspeicher übernommen.

Dieses Verfahren hat gegenüber dem Painter's Algorithmus den Vorteil, daß der z-Wert je Pixel und nicht je Polygon betrachtet wird. Somit können auch sich durchdringende Polygone korrekt dargestellt werden.

Transparenz

Grafikbibiliotheken wie OpenGL, auf der Java3D aufbaut, unterstützen die Darstellung halbtransparenter Objekte. Dafür wird jedem Pixel ein Opazitätswert α zugeordnet, der angibt, wie transparent ein Objekt dargestellt wird. Bei $\alpha = 0$ ist das Objekt völlig transparent und bei $\alpha = 1$ ist es völlig opak.

Überlappen sich nun zwei Polygone in der Projektionsebene, so wird der Farbwert der Mischfarbe aus den beiden Polygonfarbwerten und den beiden Opazitätswerten berechnet. Dabei wird die Farbe nach folgender Formel berechnet:

$$Farbe = \alpha * Vordergrundfarbe + (1 - \alpha) * Hintergrundfarbe$$

So wird z. B. eine gelbe Glasscheibe vor rotem Hintergrund im Überlappungsbereich Orange erscheinen. Der resultierende z-Wert ist der des vorderen Polygons. Deshalb müssen die halbtransparenten Flächen auch von hinten nach vorne berechnet werden. Es ist also eine Sortierung der halbtransparenten Flächen wie beim Painter's Algorithmus nötig.

Ray-Tracing (Strahlenverfolgung)

Beim Ray-Tracing werden, im Gegensatz zu den direkten Schattierungsverfahren von oben, die Pixel des Bildraumes erst in den Objektraum transformiert, wo die Beleuchtungsphänomene geometrisch und teilweise auch physikalisch genau berechnet werden können. Ausgehend vom Bildraster in der Projektionsebene erzeugt man durch jeden Pixel des Rasters einen Strahl vom Augenpunkt (Ursprung des Kamerakoordinatensystems) wie in Abb. 11.3 dargestellt. Für jeden Strahl wird der erste Schnittpunkt mit einem Objekt berechnet . Durch Bestimmen der Normalenrichtung der Oberfläche, des Lichteinfallswinkels und des Winkels des Betrachters gegenüber der Normalen kann dann zusammen mit den Materialeigenschaften die diffuse und spekuläre Lichtreflexion nach dem Phong-Modell berechnet werden.

Rekursives Ray-Tracing

Die eigentlichen Vorteile des Ray-Tracing kommen erst zum Vorschein, wenn man es rekursiv ausführt. Dabei werden mehrfach spekulär-spekuläre Lichtreflexionen durch ideale Spiegelung angenähert. Jeder Strahl der auf ein Objekt auftrifft, wird also durch Reflexion weitergeleitet und kann noch auf mehrere andere Objekte treffen. Dabei schwächt der Strahl allerdings mit der Zeit ab. Jedoch unterstützt Java3D dies nicht.

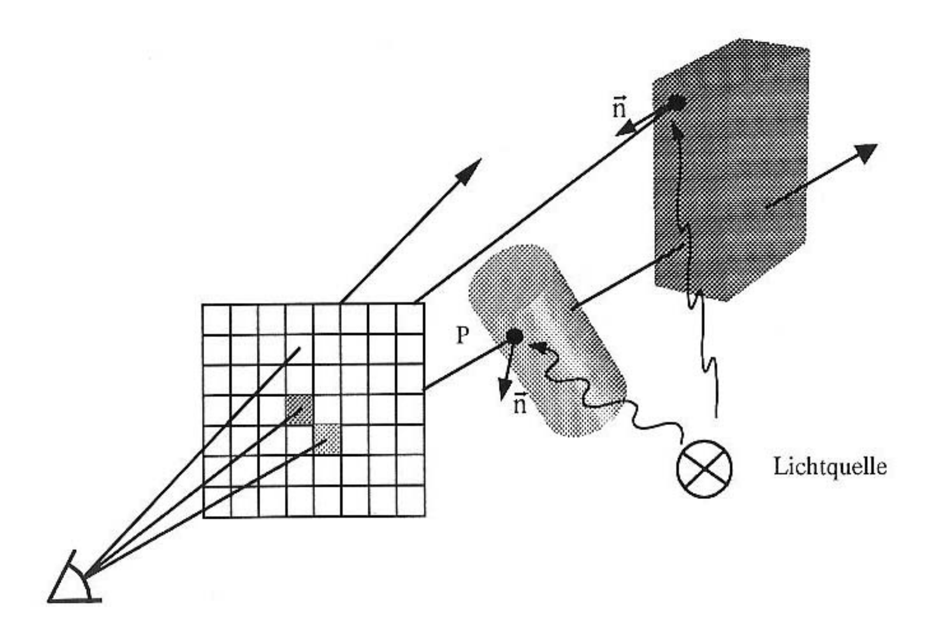


Abbildung 11.3.: RayTracing (Brüderlein und Meier, 2000)

Textur-Mapping und Texturen

Beim Textur-Mapping wird ein 2D-Rasterbild auf ein 3D-Modell appliziert. Dies ist eine der häufigsten Anwendungen der Computergrafik. Dabei unterscheidet man zwischen berechneten Texturen und Texturen von realen Oberflächen, die als Rastergrafik vorliegen.

Bei der Definition des Textur-Mapping unterscheidet man zwischen parametrischen Texturen, bei denen ein Rasterbild auf ein 3D-Polygon gelegt wird, indem man den Eckpunkten des Polygons Texturkoordinaten zuordnet wie in Abb. 11.4, und projezierte Texturen, bei denen die Texturen wie bei einem Diaprojektor auf die 3D-Objekte projiziert werden.

Affines Textur-Mapping

Das affine Textur-Mapping ist ein einfaches parametrisches Verfahren, eine Textur aus dem Textur-Raum in den Bild-Raum auf ein Objekt zu übertragen. Dabei werden, ohne Beachtung der Perspektive, den Eckpunkten eines frei gewählten Dreiecks im Bildraum entsprechende Texturkoordinaten, also auch die Eckpunkte eines Dreiecks, diesmal aber auf dem Objekt, zugeordnet, wie es in Abb. 11.4 dargestellt wird. Die Zuteilung der einzelnen Pixel erfolgt dann über eine lineare Interpolation.

Perspektivisches Textur-Mapping

Ein weiteres parametrisches Verfahren, das im Gegensatz zum affinen Textur-Mapping mit einem Zwischenschritt arbeitet, ist das Perspektivische Textur-Mapping. Dabei werden zuerst die Objektraumkoordinaten, also die dreidimensionalen Koordinaten des Objektes bestimmt,

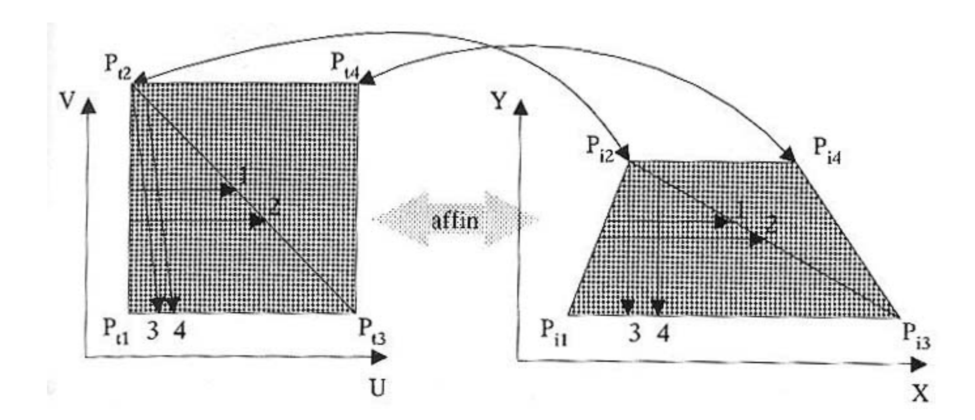


Abbildung 11.4.: Affines Mapping (Brüderlein und Meier, 2000)

bevor sie in den zweidimensionalen Bildraum transformiert werden, wie in Abb. 11.5 dargestellt. Dadurch liefert diese Methode auch bei perspektivischer Ansicht geometrisch korrekte Bilder. Sie ist allerdings mit höherem Berechnungsaufwand verbunden.

Projezierte Texturen

In dieser Texturart werden die Texturkoordinaten der projizierten Texturen aus der aktuellen Position der einzelnen Polygone im Raum berechnet. Es entspricht, wie in Abschnitt 11.2.2 erwähnt, der Projektion des Bildes eines Diaprojektors auf ein Objekt. Dabei kann man unterscheiden, ob die Position relativ zum Projektorstandpunkt oder relativ zu den Objektkoordinaten betrachtet werden soll. Falls ersteres der Fall ist, können sich die Objekte relativ zur Textur bewegen. Wird die Textur relativ zu den Objektkoordinaten betrachtet, so bewegt sich die Textur mit dem Objekt mit, wie es in Abb. 11.6 dargestellt ist.

Mip-Mapping

In einer Szene aus Objekten mit derselben Textur können sich diese Objekte in unterschiedlichen Entfernungen zum Betrachter befinden. Um die Texturen für jede Entfernung nicht neu berechnen zu müssen, benutzt man Mip-Mapping. Dabei wird vor der eigentlichen Verwendung der Texturen für verschiedene Entfernungen die Textur berechnet, und die Ergebnisse gespeichert. Dabei wird lediglich ein Drittel mehr Speicherplatz gebraucht. Die Erzeugung der Mip Map muss nur einmal stattfinden, da sie danach gespeichert werden kann. Bei der Darstellung eines Polygons wird dann nur noch die passende Auflösungsstufe ausgewählt und die Pixel zwischen den Textur-Eckkoordinaten der gewählten Auflösung auf das Polygon im Bildraum transformiert.

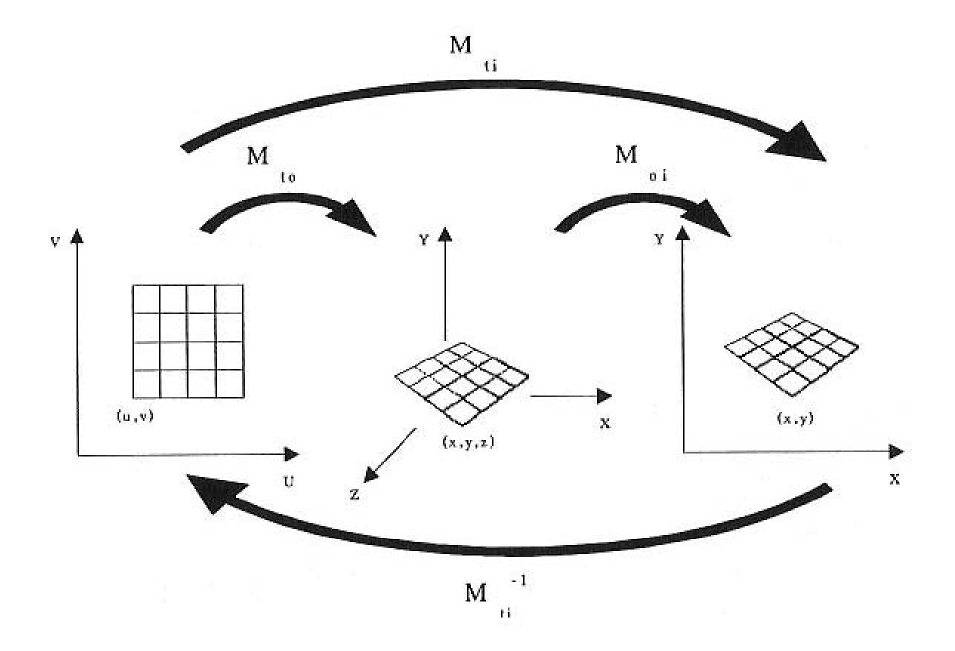


Abbildung 11.5.: Perspektivisches Textur-Mapping (Brüderlein und Meier, 2000)

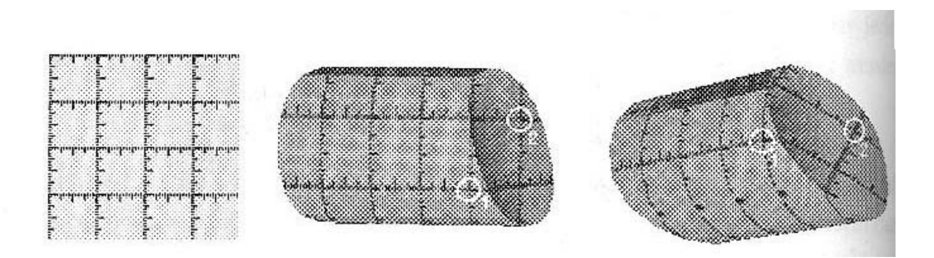


Abbildung 11.6.: Textur und Körper mit projizierter Textur (Brüderlein und Meier, 2000)

11.3 Einführung in Java 3D

Zuerst wird eine allgemeine Einführung in Java3D gegeben und anschließend auf deren Klassenbibliotheken eingegangen.

11.3.1 Allgemeines zu Java3D

Java3D ist eine optionale Erweiterung der Java2 Plattform. Veröffentlicht wurde Java3D 1.0 1998; zur Zeit (September 2003) ist die Version 1.3.1 aktuell. Somit stellt es sowohl ein ausgereiftes Produkt dar, als auch ein modernes im Vergleich mit OpenGL, welches Anfang der 90er Jahre erschienen ist. Das Java3D Application Programming Interface dient zur Erstellung, Darstellung und Animation von dreidimensionalen Szenen.

Ferner nutzt Java3D die Hardwarebeschleunigung aus, da es auf einer der beiden folgenden Low-Level APIs basiert: entweder OpenGL (SGI) oder DirectX (Microsoft), je nach der Java3D Version, die installiert wurde. Somit stellt Java3D einen plattformunabhängigen Wrapper um diese Bibliothek dar.

Java3D füllt die Lücke zwischen Low-Level APIs wie OpenGL und High-Level APIs wie VRML. Prinzipiell basiert Java3D auf asynchrone Threads, die unabhängig arbeiten (z.B. Eingabe-Management, Sound-Berechnung, Graphik-Rendering etc.).

11.3.2 Die Klassenbibliothek

Im folgenden werden einige wesentlichen Klassen der Java3D Klassenbibliothek näher erläutert.

- **Mathematik in Java 3D**
  Für die in Kapitel 11.2 oben erwähnten mathematischen Berechnungen wie Matrix und Vektoroperationen stellt Java das Package **„javax.vecmath“** bereit.
  Beispiele für Klassen wären Tupel3f, Matrix3d, Color3f.

- **Grundkörper**
  Einige vordefiniert Grundkörper können mit den Klassen des Paketes **„com.sun.j3d.utils.geometry“** erstellt werden. Wie zum Beispiel Kugeln, Würfel, Kegel oder Zylinder.

- **Interaktionen**
  Ein Paket zur Reaktion der Anwendung auf Benutzereingaben durch etwa Tastatur oder Maus stellt das Paket **„com.sun.j3d.utils.behaviors.*“** bereit.

- **Organisation des virtuellen Universums in Java3D**
  Das virtuelle Universum ist der Raum, in dem die gesamte Szenerie dargestellt werden soll. Die Handhabung der Elemente, Lichteffekte, Bewegungen wie auch der Betrachtungspunkt und die Zeichenfläche oder die Bildschirmausgabe wird durch das Paket: **„javax.media.j3d.*“** bereitgestellt. Dort befinden sich die wesentlichen Klassen, die das Grundkonzept von Java realisieren: der Szenegraph. Auf diesen soll im folgenden

Abschnitt näher eingegangen werden.

11.4 Der Scenegraph

In diesem Kapitel wird der Szenegraph erläutert, das wesentliche Konzept in Java3D. Mit dessen Hilfe wird auch die Realisierung einiger Java3D Konstrukte veranschaulicht. Ferner werden in diesem Kapitel zahlreiche englische Fachtermini eingeführt. Der Grund liegt darin, dass die Literatur zu Java3D sowohl im Internet als auch in Buchform, fast ausschließlig in englischer Sprache zu finden ist, und so der Leser mit den Begriffen bei eigener Recherche schon vertraut ist.

11.4.1 Grundlagen des Szenegraphen

Ein Szenegraph ist eine hierarchische Datenstruktur in Form eines gerichteten azyklischen Graphen. Grob lässt sich sagen, der Szenegraph spezifiziert den Inhalt der gerendert werden soll. In ihm werden alle graphischen Objekte zusammen mit den Betrachtereigenschaften gespeichert. Java3D rendert den Szenegraph am Bildschirm, und sorgt für eine korrekte perspektivische Darstellung und ist auch für die Renderingreihenfolge zuständig.

11.4.2 Basiselemente des Szenegraphen

Ein Szenegraph besteht aus verschiedenen Elementen die im Folgenden beschrieben werden und deren Zusammenspiel erläutert wird. Eine 3-dimensionale virtuelle Welt mit all ihren geometrischen Objekten wird in Java3D als `VirtualUniverse` bezeichnet. Dies stellt die Wurzel eines jeden Szenegraphen dar und sorgt für eine Kapselung für den Fall, dass mehrere Szenegraphen gleichzeitig vorhanden sind. Die Knoten an sich in einem Szenegraphen stellen instanziierte Objekte dar. Wenn Objekte an den Szenegraphen angehangen werden, so nennt man es in Java3d „they become 'live'“. In Java3D wird zwischen Knoten mit Kindern, den so genannten `GroupNodes` und den Knoten ohne Kindern also den Blättern, auch `LeafNodes` genannt, unterschieden. Beides stellen abstrakte Oberklassen dar, welche aus der Klasse `node` abgeleitet wurden.

GroupNodes

`GroupNodes` haben genau einen Elternknoten und abzählbar viele Kindknoten, wobei diese dann ebenfalls wieder `GroupNodes` oder Blätter sein können. Die folgende Abbildung 11.7 soll dies verdeutlichen. Ferner werden hier zwei wesentliche Knotentypen eingeführt, die von der Klasse der zuvor erwähnten `GroupNodes` abgeleitet werden. Dies sind die `BranchGroups` und `TransformGroups`. Erstere stellen die Wurzel von weiteren eigenständigen Teilgraphen dar. Wenn Elemente eines Szenegraphen zur Laufzeit entfernt werden sollen, kann dies nur über die entsprechende `BranchGroup` geschehen. Mit dem Methodenaufruf `test1.detach()` wird die `BranchGroup` mit dem Namen `test1` und allen an ihr befindlichen Kindern aus dem Szenegraphen entfernt.

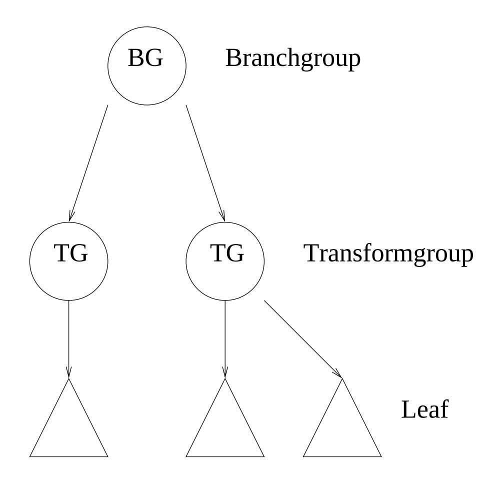


Abbildung 11.7.: Auschnitt aus einem Szenegraphen

Eine zweite wichtige Klasse von `GroupNodes` sind die `TransformGroups`. Diese beinhalten Verschiebungs-, Rotations- und Skalierungsinformationen, die an all ihre Kinder weitergeben werden. Wird also eine Transformation auf die `TransformGroup` im rechten Zweig angewendet, so wirkt sich diese auf beide Blätter aus.

Weitere `GroupNodes` sind nicht so wesentlich und werden daher nur am Rande erwähnt, wie etwa `OrderdGroup` und `Switch` mit dessen Hilfe die Renderingreihenfolge der Objekte beeinflusst werden kann und die `SharedGroup`, die die Möglichkeit bietet, Subgraphen an verschiedenen Stellen im Szenegraphen zu teilen.

LeafNodes

Die eigentlichen gegenständlichen Objekte einer dreidimensionalen Szene, wie zum Beispiel geometrische Grundkörper aber auch der Szenenhintergrund, Lichteffekte, Sounds und Benutzerinteraktionen, werden in den Blättern des Szenegraphen, den `LeafNodes`, gespeichert. In Java3D werden zwei Begriffe zur Klassifizierung von Blättern benutzt. Zum einen sind dies `ShapeNodes`, welche die Informationen der dreidimensionalen Objekte beinhalten und zum anderen die `EnvironmentNodes`, welche die Darstellung in Form von Licht, Hintergrund und Benutzerinteraktionen beeinflussen.

Knotenkomponenten

Ein gegenständliches Gebilde besitzt zum einen eine Geometrie, zum Beispiel für einen zwölfseitigen Würfel eine spezielle Anordnung der Eckpunkte, zum anderen Informationen über ihre Erscheinung, wie Farbe oder etwa Reflexionsverhalten des Materials. Die Geometriedaten und die Erscheinungsinformationen werden in den sogenannten `NodeComponents` gespeichert. Nun könnte zum Beispiel Information über die Geometrie eines zwölfseitigen Würfels in einer Knotenkomponente gespeichert werden, die dann für zwei Blätter benutzt werden soll. Während beide Blätter eine eigene Textur auf den selben geometrischen Kör-

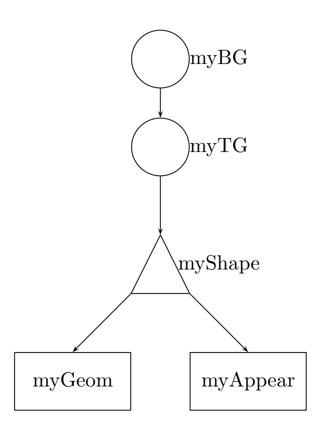


Abbildung 11.8.: Der zu implementierende Teilgraph

per legen, also beide eine eigene Knotenkomponente für die Erscheinung (`appearance NodeComponent`) besitzen, benutzen aber beide die selbe Geometrieknotenkomponente.

11.4.3 Konstruktion eines Teilgraphen

Im Folgenden wird nun die Realisierung eines Teilgrafen in Java3D erläutert. Dabei wird der in Abbildung 11.8 gezeigte Teilgraph erzeugt.

```
Appearance myAppear = new Appearance();
Geometry myGeom = new Geometry();
Shape3D myShape = new Shape3D(myAppear, myGeom);
TransformGroup myTG = new TransformGroup();
5 myTG.addChild(myShape);

BranchGroup myBG = new BranchGroup();
myBG.addChild (myTG);

10 myTG.compile();
```

In Zeile 1 und 2 werden zwei Knotenkomponenten instanziert, in Zeile 4 und 6 werden die beiden Knotenobjekte erzeugt. Bei der Instanziierung des Blattes in Zeile 3 werden die beiden Knotenkomponenten referenziert. Da die beiden Klassen der Objekte myTG und myBG von der gemeinsamen Oberklasse `Group` erben, besitzen beide die Methode `addChild(myNode)`, die den übergebenen Knoten `myNode` mit all seinen Kindknoten an das aufrufende Objekt hängt (Zeile 5 und 7). Die letzte Zeile dient eines schnelleren Renderings. Dieser Aufruf ist nur bei `BranchGroups` möglich.

11.4.4 Ein kompletter Szenegraph

Eine ganz wesentliche Klasse, die zur Erstellung eines kompletten Szenegraphen benötigt wird, ist die Klasse `Locale`. Diese definiert eine geographische Region innerhalb des virtu-

ellen Universums. Über ein weiteres Objekt, dem `HiResCoord`, wird für die `Locale` der Ursprung, die Größe und die Skalierung des Koordinatensystems festgelegt. Da die `Locale` in direktem Bezug zu dem virtuellen Universum liegt, ist sie im Szenegraphen direkt unter dem `VirtualUniverse` angeordnet (siehe Abbildung 11.9). Dies geschieht automatisch bei der Instanziierung einer `Locale`, da ihr zwingend ein `VirtualUniverse` als Parameter übergeben werden muss, ihr also gesagt werden muss, in welchem Universum sie das Gebiet aufspannen soll. Wichtig ist weiterhin, dass die `Locale` ein Sammelkontainer für Subgraphen ist. Das heisst, an ihr können beliebig viele `BranchGroups` angehangen werden.

Für eine übersichtliche Strukturierung wird der Szenegraph in zwei verschiedene Bereiche unterteilt. Das ist zum einen der so genannte `ContentBranch` und zum anderen der `Viewbranch`. Diese Trennung findet an der im obigen Abschnitt erläuterten `Locale` statt, wie in Abbildung 11.9 zu sehen ist. Deshalb besitzt sie sinvollerweise immer mindestens zwei `BranchGroups`. Einer dieser Subgraphen stellt den `ViewBranch` dar, die restlichen stellen den `ContentBranch` dar. Konkret kann man als Beispiel für einen `ContentBranch` den oben in Abbildung 11.8 erzeugten Teilgraphen nehmen, der an die `Locale` angehangen wird. Dies geschieht mit `myLocale.addBranchGroup(myBG);`

Im Folgenden wird nun der `ViewBranch` näher beschrieben. Prinzipiell befasst sich dieser mit der Ausgabe der Szenerie auf einen Bildschirm. Er endet in dem Blatt der `ViewPlattform`. Diese ist so etwas wie eine virtuelle Plattform, auf die der Benutzer steht, sich die Szenerie anschaut und sich in ihr bewegt. Wenn der Benutzer beispielsweise seine Blickrichtung auf die Szenerie ändert, geschieht dies mit Hilfe der direkt übergeordneten `TransformGroup`. Wenn in einer Szenerie zwischen mehreren vorgegebenen Kamerapositionen gewechselt werden soll, so kann dies mit Hilfe mehrerer `ViewPlattformen` realisiert werden.

Dem `View`-Objekt wird eine `ViewPlattform` zugeordnet und faßt nun alle zur Projektion des aktuellen Bildes notwendigen Informationen zusammen. Die so gerenderte Szene wird dann auf dem `Canvas3D`, einer GUI-Komponente mit einem darin integrierten Darstellungsfenster, projiziert und in der Regel über den Bildschirm ausgegeben.

11.5 Realisierung einiger Problemstellungen

In diesem Kapitel wird die Realisierung einiger einfacher Anwendungsbeispiele erläutert und deren Implementierung dargestellt. Ferner wird dabei auch auf spezielle weitere Eigenschaften in Java3D eingegangen. Auf Grund der imensen Komplexität der `Behaviors` werden diese nur sehr oberflächlich behandelt, können aber wegen ihrer Wichtigkeit nicht weggelassen werden.

11.5.1 Verschieben eines Gegenstands

Zum Verschieben eines Gegenstands trägt im wesentlichen die Klasse `Transform3D` bei. Diese wird direkt von `java.lang.object` abgeleitet. Deren Objekte sind somit eigenständig und vom Szenegraphen unabhängig. In diesem Objekt wird im allgemeinen Fall, die, wie in Abschnitt 2.1.3 beschrieben, für Transformationen benötigte 4x4 Matrix gespeichert. Die Matrix kann dem `Transform3d`-Objekt bei der Instanziierung in Form eines Arrays mit 16 Elementen übergeben werden.

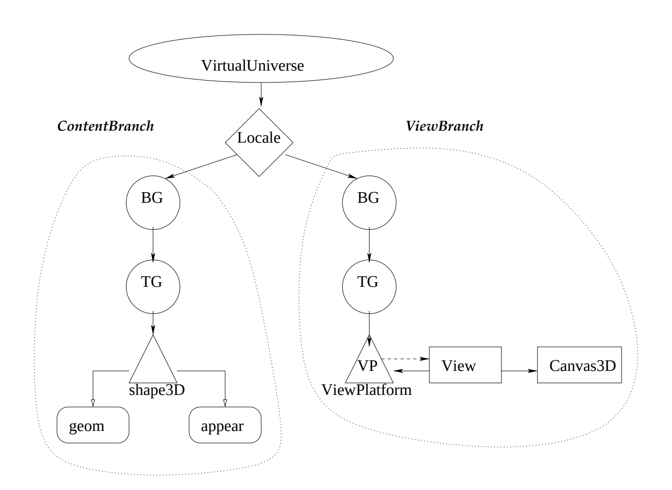


Abbildung 11.9.: Der Szenegraphen eines kompletten Programms

Um nun einen Gegenstand oder ähnliches Element (Lichtquelle oder Schallquelle) zu verschieben, muss die Transformation der `TransformGroup` mitgeteilt werden, an dessen Kindknoten sich das entsprechende zu verschiebende Objekt im Szenegraphen befindet. Dazu bietet die Klasse `TransformGroup` die Methode `setTransform(...)`. Dieser wird das `Transform3d`-Objekt übergeben. Alternativ kann bei Erzeugung einer `TransformGroup` direkt eine Transformation übergeben werden. Falls dies nicht geschieht, wird bei der Instanzierung die Einheitsmatrix erzeugt. Das heisst, dass alle Objekte, die an die `TransformGroup` angehängt werden, noch im Ursprung dargestellt werden.
Es soll nochmals darauf hingeweisen werden, dass eine Transformation sich auf alle vorhanden und künftigen Kindknoten der entsprechenden `TransformGroup` auswirkt.

11.5.2 Erzeugung einer Lichtquelle

In das Java3D Universum können verschieden Arten von Lichtquellen hinzugefügt werden, wie z.B. ambientes Licht, direkt strahlendes Licht oder punktförmige Lichtquellen, vergleiche hierzu Kapitel 11.2.2. Lichtquellen werden in Java durch `LeafNodes` repräsentiert, die von der abstrakten Klasse `light` abgeleitet werden.
Um zum Beispiel eine Punktförmige Lichtquelle wie eine Kerze, die in alle Richtungen gleiches Licht emittiert, zu erzeugen, wird der Konstruktor der Klasse `PointLight` aufgerufen:

```
PointLight candle = new PointLight(on, white, position, myatt);
```

Dabei werden die einzelnen Parameter im folgenden kurz erläutert.

- Der Parameter `on` vom Typ `boolean` legt fest, ob die Lichtquelle eingeschaltet also aktiv ist oder nicht.

- Der Parameter `white` ist vom Typ `Color3f`. Dies ist ein Farbobjekt welches nach dem RGB-Modell, siehe hierzu Kapitel 11.2.2, erzeugt wurde und gibt die Farbe des emittierten Licht an. Farbobjekte sind Bestandteil des Packetes `javax.vecmath.*`.

- Der Parameter `position` ist vom Typ `Point3f` und legt die Position der Lichtquelle fest.

- Der Parameter `myatt` ist wie der letzte auch ein 3er Tupel. Seine Elemente legen die Stärke der Dämpfung fest.

11.5.3 Erzeugen einer Geometrie

Wie in dem Abschnitt 11.4.2 beschrieben, ist zur Erstellung einer 3-dimensionalen Form (eines `Shape3D`) eine spezielle Knotenkomponente notwendig, die `Geometry`-Klasse. Mit dieser Klasse wird die Topologie und Geometrie festgelegt, die zwingend von der 3D-Form benötigt werden. Um ein grobes Gefühl dafür zu bekommen, soll in dem folgenden Beispiel ein Würfeldrahtgitter erzeugt werden.
Dazu wird ein spezielles `GeometryArray` benutzt. Dies stellt eine Ansammlung von Punkten dar, die je nach Art des `GeometryArray` auf unterschiedlicher Weise verbunden werden. Für einen Würfel bietet sich das `QuadArray` an, welches 4 Punkte zu einem Viereck verbindet. Zur Verdeutlichung nochmal die Vererbungshierarchie:

```
NodeComponent -> Geometry -> GeometryArray -> QuaddArray

1 Point3d myCoords = new Point3d[24];
2 myCoords[0] = new Point3d(1.0,1.0,-1.0)
...
3 QuadArray myGeom = new QuadArray(myCoords.length,
  QuadArray.COORDINATES);
4 myGeom.setCoordinates(0,myCoords)
5 Shape3D myCube = new Shape3D();
6 myCube.setGeometry(myGeom);
```

Ein Würfel besteht aus 6 Seiten und jede dieser Seiten hat 4 Eckpunkte. Somit muss ein Feld mit 24 Punkten erzeugt werden.
Anschließend wird in Zeile 3 ein `QuadArray` Objekt instanziiert. Nun wird dem Objekt in Zeile 4 das eigentliche Feld übergeben und mitgeteilt, welches der Startknoten in dem Feld ist, in diesem Fall also Element 0. Diese fertige Geometrie muss nun noch dem `Shape3D` Objekt übergeben werden (Zeile 6).

11.5.4 Interaktionen

Benutzer-Interaktionen und Animationen innerhalb des virtuellen Universums werden über sogenannte `Behaviors` gesteuert. Zur Einordnung in den Szenegraph dazu die Vererbungshierarchie:

```
Node -> Leaf -> Behavior
```

Behaviors können über Ereignisse und Bedingungen aktiviert und deaktivieret werden. Dies ermöglicht eine Verschiebung der Szene zur Laufzeit. Eine wichtige abgeleitete Klasse ist der `Interpolator`. Dieser führt über einen bestimmten Zeitraum eine Bewegung aus. Die Transformation des Objekts geschieht durch das `Behavior` prinzipiell wie in Abschnitt 11.5.1 beschrieben. Meist reicht die Angabe von Start- und Endpunkt des zu bewegenden Objekts aus, der Weg dazwischen wird automatisch interpoliert.
Für den Umgang mit `Behaviors` spielen die folgenden drei Methoden eine wichtige Rolle:

- `void initialize();`
  diese Methode wird aufgerufen, wenn die `Branchgroup`, an der das `Behavior` hängt, an die `Locale` gesetzt wird. In ihr wird als letztes die Methode `void wakeupOn();` aufgerufen.
- `void processStimulus(java.util.Enumeration criteria);`
  diese Methode wird aufgerufen, wenn ein Ereigniss stattfand, daß das `Behavior` geweckt hat.
- `void wakeupOn();`
  in dieser Methode steht, welche Ereignisse das `Behavior` aufwachen lassen sollen.

Ferner ist noch anzumerken, dass in Java3D schon zahlreiche vorgefertigte `Behaviors` vorhanden sind, wie etwa das `KeyNavigatorBehavior` zur Navigation im Raum, das `DistanceLOD` für die Detailtiefe von Objekten oder das `OrbitBehavior` zur Interaktion mit Gegenständen.

11.6 Fazit

Java3D ist ein mächtiges Werkzeug für die Erzeugung von 3D-Objekten und ihre Interaktion. Es ist so mächtig, daß es, um es vorzustellen, eine ganze Vorlesung füllen würde. Deshalb wurde hier Java3D auch nur ansatzweise und sicherlich unvollständig vorgestellt. Wir hoffen jedoch, daß hiermit ein kleiner Einstieg in die Java3D Welt möglich ist.

Wenn man die Struktur hinter Java3D erst einmal verstanden hat, ist sie einfach und flexibel zu handhaben. Alles in allem ist Java3D ein durchdachtes Produkt, daß wir wahrscheinlich bald sowohl verfluchen als auch zu schätzen wissen werden.

Dreidimensionale Visualisierungstechniken

Michael Nöthe

12.1 Einführung

In dieser Ausarbeitung geht es um Konzepte und Techniken zur dreidimensionalen Visualisierung von Software. Insbesondere wird dabei auf die im 3-D Klassenbrowser für Java (im folgenden kurz J3Browser) von Frank Engelen (Engelen, 2000) verwendeten Visualisierungstechniken eingegangen.

Im weiteren Verlauf dieses ersten Kapitels werden zunächst einige Grundlagen zum Themenbereich der allgemeinen Visualisierung erläutert. Das Kapitel 12.2 beschäftigt sich mit konkreten Techniken zur Darstellung von Informationen, wobei jeweils kurz diskutiert wird, für welche Strukturierung sich eine Technik eignet und was ihre Vor- und Nachteile sind. Kapitel 12.3 schließlich stellt den J3Browser vor und beschreibt, wie die zuvor aufgeführten Techniken verwendet werden, um eine Visualisierung zu erstellen.

12.1.1 Grundlagen

In diesem Unterkapitel soll der Begriff „Visualisierung“ genauer betrachtet werden. Dazu wird zunächst anhand einer Definition kurz erläutert, was im Zusammenhang dieser Arbeit unter Visualisierung zu verstehen ist. Anschließend wird der Einsatzbereich festgelegt, für den die hier vorgestellten Darstellungstechniken verwendet werden.

Begriffsklärung

Der Begriff „Visualisierung“ wird in (UseNet-Gruppe, 2004) folgendermaßen definiert :

> Visualization is the use of computer-generated media based on data in the service of human insight/learning.

Die Aufgabe einer Visualisierung besteht also darin, Daten so darzustellen, dass sie für einen Betrachter gut und schnell zu erfassen sind. Unter dem Begriff „visualisieren“ ist daher zunächst nichts anderes zu verstehen als „[Daten] für das Auge gefällig zu gestalten“ (Leisering, 1999). Ein wichtiges Qualitätsmerkmal einer Visualisierung ist demzufolge ihre Expressivität, also ihre Ausdrucksstärke und Übersichtlichkeit (Reichenberger und Steinmetz, 1999).

Einsatzbereiche für Visualisierungen

Es existieren verschiedene Anwendungsbereiche für Visualisierungen, z.B.die wissenschaftliche Visualisierung für die Darstellung quantitativer Daten oder die Informationsvisualisierung zur Darstellung von strukturell zusammenhängenden Daten und Informationen. Das Anwendungsfeld, das im weiteren betrachtet wird, ist die Softwarevisualisierung, speziell die Programmvisualisierung (Engelen, 2000, Seite 13).

12.1.2 Dreidimensionale Darstellungen

Da es in dieser Ausarbeitung um ein dreidimensionales Visualisierungssystem geht, werden hier zunächst einige Grundbegriffe aus dem Bereich der dreidimensionalen Graphik erklärt.

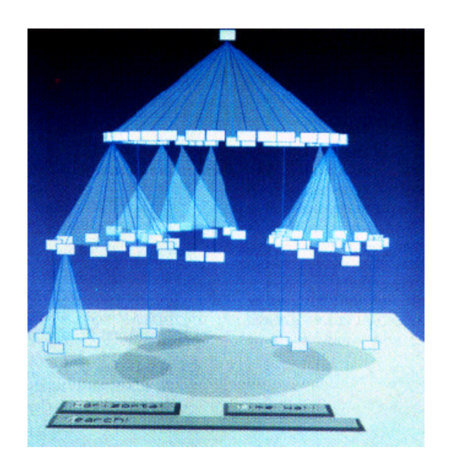
Eine dreidimensionale Graphik, auch Szene genannt, besteht aus einer Vielzahl von graphischen Objekten. Um eine solche Szene auf dem Bildschirm darzustellen, wird eine virtuelle Kamera benutzt. Diese Kamera nimmt in Abhängigkeit ihres Standpunktes und Blickwinkels ein Bild der Szene auf und gibt es auf dem Bildschirm wieder. Durch eine Veränderung der beiden Kameraparameter Standpunkt und Blickwinkel kann ein Benutzer in dieser Szene navigieren. Diese Möglichkeit der „Bewegung" durch die Szene trägt dazu bei, beim Benutzer den Eindruck einer räumlichen Darstellung auf der zweidimensionalen Bildschirmfläche zu erzeugen.

12.2 Überblick über die vorhandenen Visualisierungstechniken

In diesem Kapitel werden Darstellungstechniken vorgestellt, die im J3Browser verwendet werden. Die einzelnen Techniken werden kurz beschrieben und – soweit möglich – anhand wesentlicher Merkmale verglichen. Wichtige Unterscheidungskriterien dieser Methoden sind die Struktur der Daten, die damit visualisiert werden kann, die Dynamik der Visualisierung und die Möglichkeit, Teilbereiche der Darstellung zu fokussieren. Es werden zunächst Techniken vorgestellt, mit deren Hilfe hierarchische Strukturen dargestellt werden können, danach geht es um Methoden für die Visualisierung allgemeinstrukturierter Daten. Im letzten Unterkapitel werden Techniken betrachtet, die nicht auf eine bestimmte Zusammenhangsstruktur beschränkt sind, sondern sich allgemeiner mit dem sogenannten Fokus- und Kontextproblem befassen.

12.2.1 Techniken zur Darstellung von Hierarchien

In diesem Unterkapitel geht es um Techniken zur Darstellung baumartiger Strukturen. Bei der Visualisierung von Javaprogrammen ergeben sich solche strukturellen Zusammenhänge z.B. bei der Gliederung von Programmteilen in Pakete oder bei Vererbungshierarchien.



(a) Cone Tree (Robertson u. a., 1991)



(b) Information Cube

Abbildung 12.1.: Darstellung hierarchischer Strukturen

Cone Trees und Cam Trees

Der einzige Unterschied zwischen Cone und Cam Trees ist die Ausrichtung. Cone Trees sind vertikal, Cam Trees horizontal ausgerichtet. Die Aussage, die im weiteren über Cone Trees getroffen werden, gelten also entsprechend auch für Cam Trees. Diese Techniken sind geeignet, hierarchisch strukturierte Informationen dreidimensional darzustellen. Ein Cone Tree (Robertson u. a., 1991) ist wie folgt aufgebaut: Die Wurzel jedes Teilbaumes steht auf der Spitze eines halbdurchsichtigen Kegels, ihre direkten Unterknoten liegen gleichmäßig verteilt auf dem Kreis, der die Grundfläche dieses Kegels umschließt. Diese Anordnung wird rekursiv fortgesetzt, um den gesamten Baum darzustellen. Das Problem der Fokussierung eines Knotens wird bei den Cone Trees durch Animation gelöst. Soll ein Knoten näher betrachtet werden, so wird er durch Drehung einzelner Teilbäume in den Vordergrund geholt.

Laut (Robertson u. a., 1991) lassen sich mit Cone Trees Strukturen von bis zu tausend Knoten effektiv visualisieren. Ein Nachteil dieser Darstellungsform ist jedoch die Tatsache, dass der benötigte Platz -besonders bei Bäumen mit hohem Verzweigungsgrad -mit zunehmender Tiefe sehr schnell wächst.

Information Cubes

Ähnlich wie Cone Trees lassen sich mit den Information Cubes (Rekimoto und Green, 1993) hierarchische Informationen darstellen. Im Gegensatz zur Cone Tree-Technik werden hier ineinander verschachtelte Würfel (Cubes) benutzt um die einzelnen Elemente darzustellen. Das Wurzelelement der darzustellenden Hierarchie wird dabei als äußerster Würfel gezeichnet, seine Unterelemente werden als kleinere Würfel in seinem Inneren dargestellt, deren Unterelemente befinden sich wiederum in ihrem Inneren. Auch diese Anordnung wird rekursiv fortgesetzt. Die Wände der Würfel sind semitransparent, um die in einem Würfel enthaltenen

Elemente sichtbar zu machen. Die Semitransparenz hat außerdem den angenehmen Effekt, dass Würfel, die zu tief verschachtelt sind, aufgrund der Aufsummierung der Transparenzwerte praktisch automatisch ausgeblendet werden. Damit der Benutzer auch innere Würfel näher betrachten kann, besteht die Möglichkeit, in die Würfel hinein zu navigieren. Nach (Rekimoto und Green, 1993) lassen sich auch mit dieser Methode bis zu tausend Elemente in einer Visualisierung unterbringen. Ein Nachteil dieser Visualisierungstechnik besteht in der fehlenden Übersichtlichkeit bei großen und tief verschachtelten Systemen, da die in der Hierarchie oberen Elemente die unteren zu sehr verdecken. Andererseits ist diese Visualisierungstechnik sicherlich gut dafür geeignet,etwa eine Enthaltensein-Relation darzustellen, z.B. eine Java-Paketstruktur.

12.2.2 Eine Technik zur Darstellung beliebig strukturierter Daten

Neben Hierarchien müssen bei der Programmvisualisierung auch weniger stark strukturierte Zusammenhänge dargestellt werden. Diese beliebigen Graphen treten z.B. bei der Betrachtung verschiedenartiger Beziehungen zwischen den Klassen eines Java-Programms auf. Vorgestellt wird hier die Technik der Federmodelle. Für einen allgemeinen Überblick über den Bereich des Graphlayout sei auf (Battista u. a., 1994) verwiesen.

Bei der Methode der Federmodelle werden die Knoten des Graphen als Verbindungspunkte angesehen, zwischen denen Federn ”gespannt” sind. Diese Federn haben als Attribute eine Ruhelänge und eine Federkonstante und können unter Einwirkung von Zug- und Druckkräften gedehnt bzw. gestaucht werden. Mit Hilfe dieser Federn kann man einen Graph z.B. so modellieren, dass durch Kanten verbundene Knoten sich nicht allzu weit voneinander entfernen oder dass die Knoten untereinander bestimmte Abstände wahren. Nach der Modellierung wird das entstandene mechanische System in einen energetisch optimalen Zustand gebracht: Es wird also berechnet,wie sich die Knoten aufgrund der Federkräfte verteilen.

12.2.3 Das Fokus- und Kontextproblem

Das Fokus-und Kontextproblem besteht darin, eine Möglichkeit zu finden, dem Betrachter sowohl Detailinformationen als auch einen möglichst umfassenden Gesamtüberblick über das darzustellende System zu bieten. Ein Konzept zur Lösung dieses Problems stellen Fish Eye Views (FEV) nach (Furnas, 1981) dar. Es gibt zwei unterschiedliche Arten von FEVs: Verzerrungs- FEVs und Filter-FEVs. Die Verzerrungs-FEVs simulieren ein Fischaugenobjektiv. Sie stellen das Zentrum des Blickfeldes, den Fokus, stark vergrößert und die Umgebung stark verkleinert dar. Dadurch soll erreicht werden, dass mehr Kontextinformationen eines gerade fokussierten Objektes dargestellt werden können als ohne Verwendung des FEVs. Filter-FEVs hingegen blenden selektiv einige Objekte aus der Darstellung aus. Abhängig von dem Objekt, welches sich im Fokus befindet, wird für alle anderen Objekte der Degree of Interest (DOI) bestimmt. Dies ist ein Wert,der die Relevanz eines jeden Objektes angibt. Die Auswahl der in der Darstellung verbleibenden Objekte wird anhand des DOI getroffen. Liegt der DOI für ein Objekt unterhalb eines Schwellenwertes, so wird es nicht mehr angezeigt. Eine Auswahl weiterer Lösungen für das Fokusproblem ist in (Engelen, 2000) zu finden.

12.3 J3Browser

Nachdem sich die ersten beiden Kapitel mit Grundlagen und Techniken der Visualisierung befasst haben, geht es nun darum, den J3Browser vorzustellen. Die Funktion des J3Browsers besteht in der „dreidimensionale Visualisierung des Aufbaus von Java-Software“ (Engelen, 2000, Seite 47). Im weiteren Verlauf dieses Kapitels wird zunächst auf die zur Darstellung verwendete Notation eingegangen (Abschnitt 12.3.1), bevor in Abschnitt 12.3.2 die Techniken zur räumlichen Anordnung der einzelnen Darstellungskomponenten näher betrachtet werden. Abschließend werden in Abschnitt 12.3.3 Methoden zur Verbesserung der Darstellung wie FEVs und Filter sowie dynamische Aspekte der Visualisierung beschrieben.

12.3.1 Notation

Bevor auf graphische Einzelheiten der Notation eingegangen werden kann, muss zunächst geklärt werden, welche Objekte in die Darstellung aufgenommen werden. Hierbei muss man zwischen darzustellenden Sprachkonzepten und graphischen Symbolen unterscheiden. Beide werden in den folgenden Abschnitten näher betrachtet.

Darzustellende Sprachkonzepte

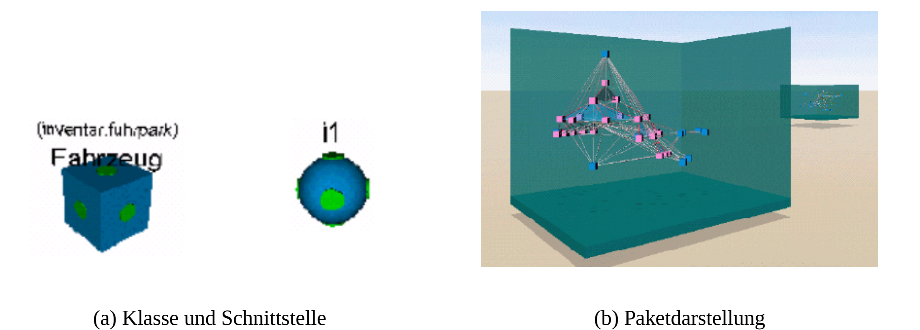
Die Sprachkonzepte,die der J3Browser verwendet, sind im wesentlichen aus der Unified Modeling Language (UML) abgeleitet. Sie lassen sich als erstes in Elemente und Beziehungen aufteilen. Element ist der Oberbegriff für Pakete und Entitäten, unter Entitäten sind schließlich Schnittstellen und Klassen zusammengefasst. Auch Beziehungen lassen sich weiter aufteilen, und zwar zunächst in Zugehörigkeiten und Verwendungen. Zugehörigkeiten können zwischen Paketen und Entitäten bestehen, und zwar in beliebiger Kombination. Die Verwendung lässt sich grob untergliedern in Import, Implementierung, Erweiterung und Benutzung. Für eine weiterführende Erläuterung der hier verwendeten Begriffe sei auf Literatur zur UML verwiesen, etwa (Booch u. a., 1999).

Verwendete graphische Symbole

Entitäten werden als kompakte, körperförmige Symbole dargestellt. Abbildung 12.2(a) zeigt die Symbole für Klassen (Quader) und Schnittstellen (Kugel), deren Form in Anlehnung an die zweidimensionalen Symbole der UML für diese Entitäten gewählt wurde, nämlich das Rechteck und den Kreis. Der Name der jeweiligen Entität wird über ihrem Symbol dargestellt. Zusätzlich haben diese Symbole sechs Eigenschaftsmarkierungen. Die Farbe der Eigenschaftsmarkierung stellt den Zugriffsmodus dar, ihre Form steht für die Instanziierbarkeit der Entität. Für genauere Angaben wird auf (Engelen, 2000) verwiesen.

Die Darstellung von Paketzugehörigkeit erfolgt mit Hilfe der Information Cube Technik. Pakete werden dabei als große Quader gezeichnet, deren Wände von außen fast völlig transparent sind (Abb. 12.2(b)). Im Inneren des Paketquaders sind die Symbole der in diesem Paket enthaltenen Entitäten angeordnet. Betrachtet man ein Paket von außen, so sieht man nur die enthaltenen Symbole und die drei rückwärtigen Wände, während die drei Vorderwände praktisch durchsichtig sind und den Blick auf das Innere freigeben.

Beziehungen zwischen Entitäten sind durch Pfeile dargestellt. Die Richtung der Beziehung



(a) Klasse und Schnittstelle (b) Paketdarstellung

Abbildung 12.2.: Symbolnotationen nach Engelen (2000)

wird sowohl durch die Spitze angedeutet, als auch durch einen Helligkeitsverlauf. Die verschiedenen Arten von Beziehungen werden durch die Farbe des Pfeils gekennzeichnet. Es wird unterschieden zwischen Benutzung (grau), Import (blau), Erweiterung (rot), Implementierung (gelb) und Enthaltensein (grün) im Sinne von inneren Elementen. Bei der Benutzungsbeziehung können außerdem noch weitere Details dargestellt werden. Für weitere Einzelheiten muss auch an dieser Stelle auf (Engelen, 2000) verwiesen werden.

12.3.2 Das Visualisierungssystem

Nachdem im Kapitel 12.2 verschiedene einzelne Visualisierungstechniken vorgestellt wurden, wird nun anhand des J3Browsers beschrieben, wie diese Techniken in einem konkreten Visualisierungssystem eingesetzt werden können. Im ersten Abschnitt wird kurz erläutert, welche Techniken zur Darstellung einzelner Beziehungen benutzt werden. Im darauffolgenden Abschnitt geht es darum, wie diese Techniken kombiniert werden, um ein Javaprogramm mit verschiedenen Beziehunge zu visualisieren.

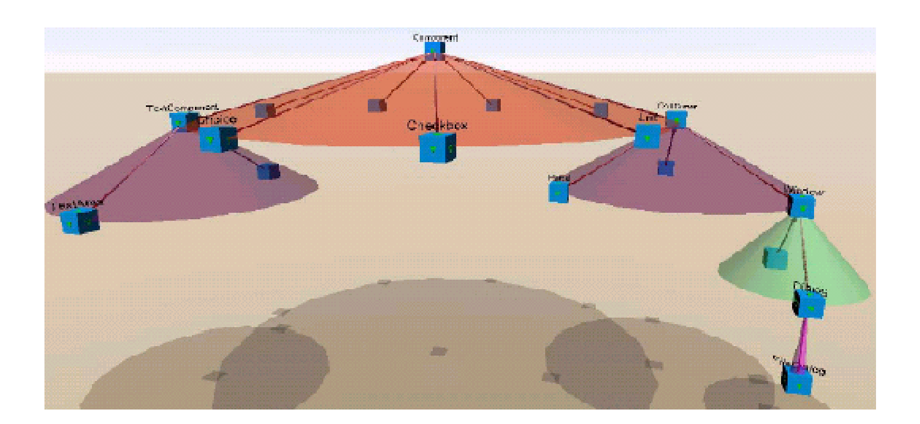
Einsatzgebiet von Einzeltechniken

Eine Möglichkeit, eine Visualisierung eines objektorientierten Programms zu erstellen, ist, es anhand der Erweiterungsstruktur seiner Klassen darzustellen. Da es in Java keine Mehrfacherbung unter Klassen gibt, ergibt sich dabei eine baumartige Struktur. Im J3Browser können für ihre Darstellung sowohl Cone Trees (Abb. 12.3(a)) als auch in einer Ebene angeordnete Bäume (Walls) (Abb. 12.3(b)) benutzt werden.

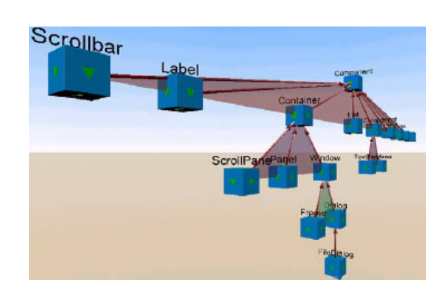
Eine weitere Möglichkeit, ein Javaprogramm zu visualisieren, ergibt sich durch die Darstellung seiner Paketstruktur. Dies geschieht im J3Browser mit Hilfe von Information Cubes (siehe Abb. 12.2(b)).

Gleichzeitige Benutzung mehrer Darstellungstechniken

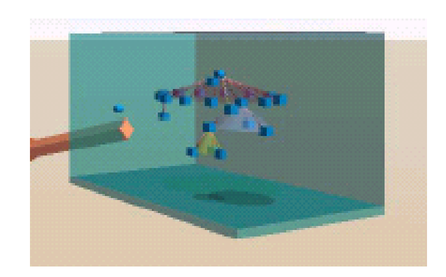
Ein Javaprogramm lässt sich auf mehrere Arten darstellen, z.B.anhand seiner Paket- oder Erweiterungsstruktur. Diese verschiedenen Möglichkeiten zeigen jedoch i.d.R. immer nur ein-



(a) als Cone Tree



(b) als Baum in der Ebene



(c) als Cone Tree in einem Information Cube

Abbildung 12.3.: Darstellung von Klassenhierarchien nach Engelen (2000)

zelne Aspekte der Programmstruktur. Um einen Gesamtüberblick zu schaffen, in den auch andere Beziehungsarten einfließen, ist es nötig, diese Techniken zu kombinieren, anzupassen und sie ggf. zu erweitern. Ein Beispiel für die Kombination von zwei Techniken ist in Abb. 12.3(c) zu sehen. In dieser Darstellung sind sowohl Paketzugehörigkeiten als auch eine Klassenhierarchie untergebracht.

Nicht in jedem Fall lassen sich verschiedene Visualisierungstechniken so problemlos wie oben in derselben Darstellung verwenden. Falls z.B. außer den Erweiterungs- auch noch Benutzungsbeziehungen dargestellt werden sollen, können sich Probleme bei der Platzierung der Beziehungspfeile ergeben. Zu den Pfeilen des Cone Tree, der in einem solchen Fall die Erweiterungshierarchie abbildet, würden Beziehungspfeile kommen, die im Inneren des Tree verlaufen. Da der Innenraum u.U. bereits mit vielen Symbolen gefüllt ist, kann es zu Überschneidungen kommen. Verstärkt wird dieses Problem noch durch die Möglichkeit, die einzelnen Kegel des Tree zu drehen, denn die Benutzungspfeile müssten immer wieder an die neue

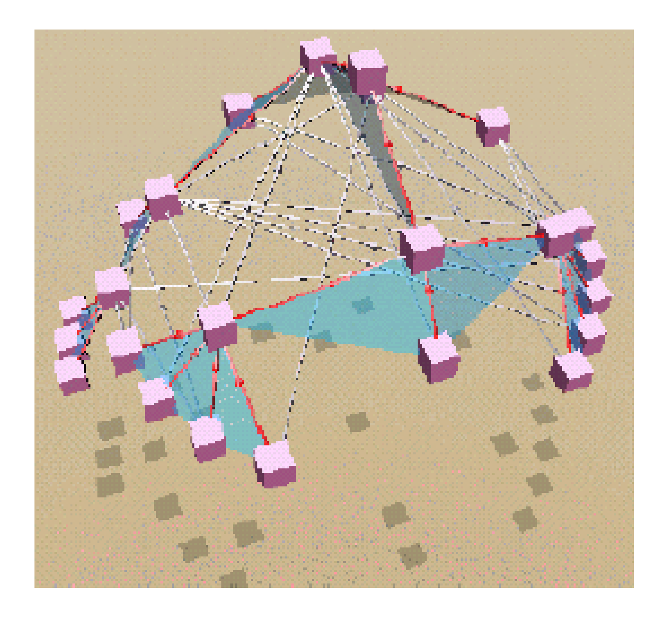


Abbildung 12.4.: Kegeldarstellung (Engelen, 2000)

Symbolanordnung angepasst werden. Abhilfe schafft hier die Kegeldarstellung, eine Variante der Cone Tree Technik. Anstatt für jeden Unterbaum einen eigenen Kegel zu benutzen, wird hier der gesamte Baum auf dem Mantel eines einzigen Kegels angeordnet. Dadurch bleibt der Innenraum dieses Gebildes frei für andere Beziehungspfeile (Abb. 12.4).

Eine weitere Möglichkeit, dem Problem der Darstellung unterschiedlicher Beziehungsstrukturen zu begegnen,besteht in der Lockerung der Regeln für die Symbolanordnung. Auf diese Weise können ebenenförmige oder auch top-down- Darstellungen entstehen. Für nähere Ausführungen zu diesen alternativen Techniken siehe (Engelen, 2000, Seite 54 ff.).

12.3.3 Techniken zur Verbesserung der Expressivität

Zum Abschluss der Vorstellung des J3Browswers sollen Methoden vorgestellt werden, mit deren Hilfe sich die Übersichtlichkeit der Darstellung erhöhen lässt. Dazu zählen u.a. Filter, eine Fokustechnik und nicht zuletzt auch die dynamischen Aspekte des vorliegenden Visualisierungssystems.

Fish Eye Views und Filter

Die gleichzeitige Anzeige aller Elemente und Beziehungen eines Programms führt i.d.R. allein aufgrund der hohen Anzahl der Symbole und Pfeile zu einer stark überladen wirkenden Darstellung. Daher sind Techniken nötig, die die Übersichtlichkeit der Darstellung gewährleisten. Eine einfache Vorgehensweise ist die Benutzung von Filtern für Beziehungspfeile. Die gezielte Ausblendung von Elementsymbolen und ihren Beziehungspfeilen ist eine andere Möglichkeit. Die dritte Technik, die angewendet wird, ist eine Abwandlung des Fish-Eye-

View. Wird ein Element fokussiert, werden alle Beziehungen, die nicht mit dem ausgewählten Element in Berührung stehen, stark transparent dargestellt.

Reduktion der Darstellungskomplexität

Außer den verschiedenen Filtertechniken gibt es im J3Browser weitere Möglichkeiten, die Zahl der darzustellenden Symbole und Pfeile zu verringern. Es ist möglich, mehrere Entitäten manuell auszuwählen und ihre Symbole durch ein einziges Gruppensymbol zu ersetzen. Eine entsprechende Methode für Pfeile fasst Beziehungen zwischen Entitäten aus verschiedenen Paketen zusammen und ersetzt deren Pfeile durch einen einzelnen Abhängigkeitspfeil zwischen jeweils zwei Paketen. Die konkreten Java-Beziehungen (Benutzung, Erweiterung, Import und Implementierung) werden dabei durch die abstrakte Abhängigkeitsbeziehung ersetzt.

Dynamische Aspekte

Die dynamischen Aspekte des J3Browsers lassen sich in zwei Klassen gliedern. Eine Klasse umfasst die Navigationsmöglichkeiten der virtuellen Kamera, also deren Bewegung in der Szene. Die andere Klasse beinhaltet die interaktive Symbolanordnung.

Die Navigationsmöglichkeiten in der Szene sind größtenteils durch den zur grafischen Darstellung verwendeten VRML-Browser (Computer Associates International, Inc. Islandia, USA, 2000) vorgegeben. Zur Steuerung der Ansicht können die zwei Achsen der Computermaus mit je einer Funktion belegt werden, so dass achsenparallele Bewegungen, Bewegungen in Blickrichtung sowie Drehungen der Blickrichtung möglich sind. Zusätzlich existiert die Funktion “Suchen”, mit deren Hilfe die virtuelle Kamera automatisch an ein angewähltes Objekt hera fährt.

Die interaktive Symbolanordnung bei Cone Trees besteht, wie in Abschnitt 2.1.1 erläutert, darin, durch Drehung von Unterbäumen ein Element in den Vordergrund zu holen. Auch für die Kegelanordnung ist diese Möglichkeit implementiert, hierbei wird der gesamte Kegel um das Wurzelelement gedreht. Dies sind die wesentlichen dynamischen Funktionen, die im J3Browser implementiert sind.

12.4 Fazit

Diese Ausarbeitung sollte einige Grundlage der Computervisualisierung vermitteln, ausgewählte Visualisierungstechniken kurz vorstellen und schließlich zeigen, wie diese Methoden in dem Visualisierungssystem J3Browser eingesetzt werden. Die benutzten Techniken haben sich bewährt, es bleibt jedoch noch Raum für Erweiterungen und Verbesserungen. Zum Abschluss soll erwähnt werden, dass hier nicht alle Funktionen des J3Browsers beschrieben werden konnten, sondern nur eine Auswahl. Für Detailinformationen sei auch hier auf die Literatur (vor allem auf Engelen (2000)) verwiesen.

KAPITEL 13

Grafische Editoren und dreidimensionale Benutzungsschnittstellen

Kai Gutberlet

13.1 Einleitung

Im Rahmen der Projektgruppe 444 des Fachbereichs Informatik der Universität Dortmund soll ein Editorframework für Entwurfsnotationen zur dreidimensionalen Darstellung und Manipulation von Softwarestrukturen erarbeitet werden. Das vorliegende Referat behandelt den Teilbereich grafische Editoren und dreidimensionale grafische Benutzungsschnittstellen mit dem Ziel, einen Überblick zu geben. Eine vertiefende Bearbeitung kann wegen des großen Themenumfangs natürlich nicht erfolgen.

13.1.1 Begriffe Editor, grafischer Editor und Benutzungsschnittstelle

Grundlage für die Diskussion über grafische Editoren und grafische Benutzungsschnittstellen ist eine klare Definition der Begriffe *Editor*, *grafischer Editor* und *Benutzungsschnittstelle*. Ein *Editor* ist laut (Rechenberg und Pomberger, 2002) ein Werkzeug zur Eingabe, Veränderung und Darstellung von Daten aller Art. Hierzu zählen zum Beispiel Text, Grafik oder Musik. *Grafische Editoren* sind spezielle Editoren, die die Bearbeitung von Daten unterstützt durch grafische Hilfsmittel ermöglichen. Dabei werden die Daten durch grafische Elemente repräsentiert und können so komfortabel bearbeitet werden. Der Begriff *Benutzungsschnittstelle* umfasst nach (Webnox Corporation, 2003) alle Bestandteile eines Rechensystems, die dem Nutzer Daten visuell, akustisch oder in anderer Form darstellen oder ihm Möglichkeiten zur Eingabe geben.

13.1.2 Diagrammeditoren und Interaktionskonzepte grafischer Editoren

Das von der PG zu entwickelnde Editorframework zielt auf Softwarestrukturen ab. Grafische Editoren, die einen objektorientierten Softwareentwurf unterstützen, lassen sich in den Bereich der Diagrammeditoren einordnen. Diagrammeditoren sind nach (Gille, 1999) grafische

Editoren, die auf eine bestimmte Diagrammsprache wie z. B. elektrische Schaltpläne oder UML zugeschnitten sind. Sie eignen sich besonders zur Visualisierung und Manipulation von Nutzdaten, die eine graphenartige Struktur besitzen.

Für die Interaktion mit grafischen Editoren existieren unter anderem die Konzepte der Struktureditoren, der syntaxgesteuerten Editoren, und der freien Editierbarkeit. Struktureditoren ermöglichen laut (Szwillus, 1990) die Erstellung und Veränderung von Diagrammen anhand korrekter Strukturen, die durch Einfügen von vordefinierten Mustern und Elementen ausgebaut werden und garantieren so syntaktische Korrektheit. Ein einfaches Beispiel verdeutlicht das Prinzip: Nach Definition ist jeder binäre Baum entweder leer oder besteht aus einer Wurzel, einem linken und einem rechten binären Unterbaum. Startend mit einem leeren Baum, kann man durch Verwendung dieses Musters jeden beliebigen binären Baum aufbauen. Das Prinzip der Struktureditoren wird in (Reps und Teitelbaum, 1989) näher erläutert. Syntaxgesteuerte Editoren überprüfen nach (Boles, 1994) bei jeder Eingabe die Richtigkeit der Konstrukte, so dass es dem Nutzer nicht möglich ist, syntaktisch falsche Grafiken zu erstellen. Um dieses Ziel zu erreichen, muss zum einen für die grafischen Elemente, welche eine visuelle Sprache bilden, eine Grammatik erstellt werden. Zum anderen müssen Datenstrukturen zur Verwaltung der komplexen syntaktischen Strukturen analog zu Syntaxbäumen aufgebaut werden. Die Realisierung der Syntaxsteuerung erfordert daher einen höheren Entwicklungsaufwand. Demgegenüber steht die einfache Möglichkeit der freien Editierbarkeit. Wie mit Zeichenprogrammen können die grafischen Elemente mit durch den Editor zur Verfügung gestellten Mitteln frei verändert werden.

Bei unserem Editorframework kann nun die Eingabe frei, syntaxgesteuert oder strukturorientiert erfolgen. Auch Kombinationen dieser Möglichkeiten sind durch Verwendung verschiedener Eingabemodi möglich. Syntaxprüfungen können aber auch explizit nach Erstellung eines Diagramms ausgeführt werden. Grafische Editoren werden heute in vielen Bereichen wie Softwareentwicklung, CAD, Netzwerkplanung und Bildbearbeitung eingesetzt.

13.1.3 Die Benutzungsschnittstelle grafischer Editoren

Die Entwicklung eines Editorframeworks zur dreidimensionalen Darstellung und Manipulation von Softwarestrukturen beinhaltet die Entwicklung einer dreidimensionalen grafischen Benutzungsschnittstelle. Dabei dürfen aber Aspekte, die für zweidimensionale grafische Benutzungsschnittstellen relevant sind, nicht außer Acht gelassen werden, denn dreidimensionale Räume sind meist in zweidimensionale grafische Benutzungsschnittstellen eingebettet. Grundelemente einer zweidimensionalen grafischen Benutzungsschnittstelle sind mausorientierte Eingabe, Fenstersteuerung, Menüführung, Buttonleisten für Standardoperationen und Iconleisten für grafische Elemente. Des Weiteren gibt es die Möglichkeit, Attribute selektierter Objekte über kontextsensitive Popupmenüs gesondert zu editieren. Grafische Benutzungsschnittstellen unterstützen den Benutzer häufig durch das WYSIWYG (What You See Is What You Get)-Konzept, bei dem er stets unmittelbar das Ergebnis seiner Eingabe auf dem Bildschirm sieht.

Bei der Gestaltung von Benutzungsschnittstellen soll nach (Rechenberg und Pomberger, 2002) und (Herrmann, 2001) auf die folgenden Ziele und Anforderungen Rücksicht genommen werden. Allgemein soll eine gute Benutzerführung es dem Nutzer möglichst einfach und komfortabel machen, seine Arbeit zu erledigen. Eine bedeutende Rolle spielt der Erlernbar-

keitsaspekt. Gute Oberflächen versetzen den Nutzer in die Lage, ohne großes Handbuchstudium oder aufwendige Schulung nach kurzer Einarbeitung mit seiner Arbeit zu beginnen und bieten zusätzlich oft gute Hilfsfunktion an. Moderne Benutzungsschnittstellen sollen auch Grundsätze der Softwareergonomie, unter der man die Anpassung der Nutzungsbedingungen an Eigenschaften der Benutzer versteht, beachten. Als wichtige Beispiele ergonomischer Grundsätze sind hier die Beachtung der Wahrnehmungspsychologie bei der Gestaltung von Bildschirmmasken, die Erwartungskonformität bei der Nutzerinteraktion und die Fehlerrobustheit der Anwendung zu nennen.

13.2 Bestandteile von dreidimensionalen grafische Benutzungsschnittstellen

Nach der Betrachtung zweidimensionaler grafischer Benutzungsschnittstellen, steht nun die Untersuchung dreidimensionaler grafischer Benutzungsschnittstellen an, die einige Ansatzpunkte für die Gestaltung der Benutzungsschnittstelle des geplanten Editors liefern kann. Der große Vorteil von dreidimensionalen grafischen Benutzungsschnittstellen ist die Möglichkeit, intuitivere Benutzungsschnittstellen erstellen zu können, da wir täglich unsere reale dreidimensionale Welt erfahren und diese Erfahrungen im Umgang mit der grafischen Benutzungsschnittstellen nutzen können. Um diese Erfahrungen nun wirklich zu nutzen, sollte die modellierte Welt die Realität nachahmen, wobei aber stets zu beachten ist, welcher Grad an Realität für den Zweck der Anwendung sinnvoll ist. Zum Beispiel kann in einer Simulation das Verschieben von Kisten nur durch Gabelstapler erlaubt sein, währenddessen dies beim Entwurf einer Lagerverwaltung äußerst ungünstig ist. So sind also stets die Anwendung und ihre Anforderungen der Ausgangspunkt für das Design der Benutzungsschnittstelle.

13.2.1 Grundlegende Begriffe für dreidimensionale grafische Benutzungsschnittstellen

Zuerst sind die für dreidimensionale Darstellungen essentielle Begriffe dreidimensionale *Welt (World)*, *Sicht (View)* und *Display* zu klären. Die dreidimensionale *Welt* stellt den Bezugsraum dar, in dem alle zu modellierenden Objekte platziert werden, und hat ihr eigenes dreidimensionales Koordinatensystem. Damit ein Nutzer etwas von der dreidimensionalen Welt sehen kann, benötigt er eine Art Stellvertreter in der dreidimensionalen Welt, der die Position und Blickrichtung seiner Augen darstellt — die *Sicht*. Sie zeigt die dreidimensionale Welt parallel oder perspektivisch projiziert auf ein zweidimensionales Display und ist in gewisser Weise vergleichbar mit einer Kamera. Außerdem besitzt sie einen eigenen dreidimensionalen Bezugsraum für Objekte in der dreidimensionalen Welt, die von ihrer Position aus zu sehen sind. Das *Display* zeigt die zweidimensionale Repräsentation der dreidimensionalen Welt aus dem Blickwinkel der zugeordneten Sicht.

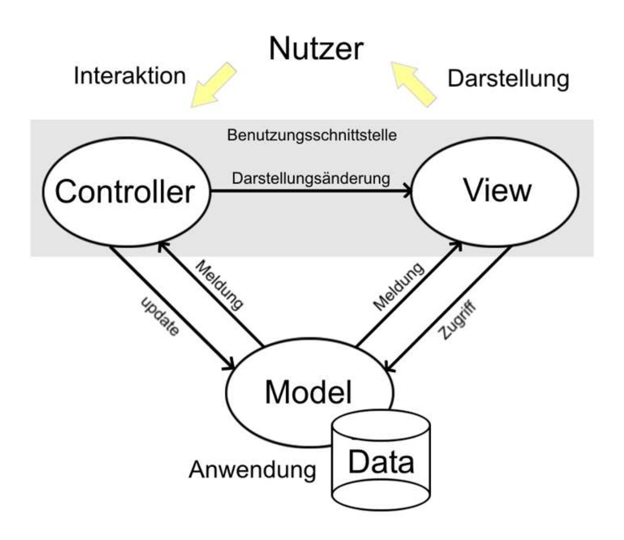


Abbildung 13.1.: Struktur des MVC-Musters nach (Kühne, 2002)

13.2.2 MVC Muster

Aktuelle grafische Benutzungsschnittstellen sind nach dem Model-View-Controller-Muster (MVC-Muster) aufgebaut, welches eine funktionale Gliederung in die drei Komponenten Modell, Sicht (View) und Steuerung (Control) vorsieht. Grundidee der Unterteilung ist nach (Kühne, 2002) die Unterscheidung zwischen anwendungsbezogenen, darstellenden und interaktiven Komponenten einer grafischen Benutzungsschnittstelle. Die Modellkomponente bildet den funktionalen Kern einer Anwendung, indem sie die Anwendungsdaten kapselt und Methoden zu deren Bearbeitung bereitstellt. Sie realisiert den kontrollierten Zugriff auf die Anwendungsdaten und sorgt für deren korrekte Aktualisierung, wozu sie insbesondere effiziente Datenstrukturen benötigt. Für die komplette grafische Darstellung ist die Sichtkomponente verantwortlich. Sie führt sowohl die Visualisierung der Daten der Modellkomponente als auch die grafische Behandlung von Eingaben, welche durch die Steuerungskomponente veranlasst wird, durch. Die grafische Eingabebehandlung bezieht sich hier auf Änderungen, die nur die Darstellung betreffen, wie z. B. Zoomen oder Scrollen in der Ansicht. Alle durch Nutzereingaben entstehenden Eingabeereignisse werden durch die Steuerungskomponente registriert und behandelt. Sie leitet Anfragen gemäß der Eingabeereignisse an die Modellkomponente (Datenänderung) oder an die Sichtkomponente (Darstellungsänderung) weiter. Dieses Zusammenspiel der Komponenten verdeutlicht Abbildung 13.1. Der große Vorteil des MVC-Musters liegt in der Trennung von Datendarstellung und Datenmodell. Der funktionale Kern der Anwendung (Modell) ist hierdurch unabhängig von der eigentlichen grafischen Benutzungsschnittstelle (Sicht und Steuerung). So lassen sich zum einen Sichten einfach austauschen, zum anderen sind mehrere Sichten in einer Anwendung leicht zu realisieren.

Das MVC-Muster bildet für grafische Benutzungsschnittstellen allgemein eine gute Entwicklungsgrundlage. Für dreidimensionale Benutzungsschnittstellen sieht (Barrilleaux, 2001)

konzeptionell eine Einteilung in die sechs Komponenten Steuerung (Control), Feedback, Visualisierung (Visualization), Navigation, Manipulation und Zugriff (Access) vor. Die Steuerungskomponente verarbeitet alle Eingaben, welche sich in der dreidimensionalen Welt zum einen auf die Veränderung der Sicht und zum anderen auf die Veränderung der Daten beziehen. Soll nun die Sicht verändert werden, wird die Funktionaltät der Navigationskomponente benötigt, da sie für die Bewegung von Sichtobjekten (Views) zuständig ist. Wenn die Darstellung der Daten in der dreidimensionalen Welt (Objects) geändert werden sollen, wird die Manipulationskomponente angesprochen. Für das Hinzufügen und Entfernen von Daten (Datas) zur dreidimensionalen Welt ist die Zugriffskomponente zuständig. Die Darstellung übernehmen die beiden Komponenten Visualisierung und Feedback. Während die Visualisierung für die Darstellung der Anwendungsdaten sorgt, verfolgt die Feedbackkomponente das Ziel, den Nutzer durch die Anwendung zu führen. Abbildung 13.2 zeigt die sechs funktionalen Komponenten.

Die Einteilung in sechs Komponenten bietet eine genauere funktionale Gliederung der dreidimensionalen Benutzungsschnittstelle und kann als Erweiterung des MVC-Musters angesehen werden. Die Steuerungskomponente des MVC-Modells ist in die Komponenten Steuerung, Navigation, Manipulation und Zugriff aufgegliedert. In den beiden Komponenten Visualisierung und Feedback findet sich die Sichtkomponente des MVC-Musters wieder. Ein Analogon zur Modellkomponente fehlt allerdings in dieser funktionalen Einteilung, da der Schwerpunkt auf der Interaktion mit dem Nutzer gelegt wurde. Die Datenobjekte (Objects) in Abbildung 13.2 weisen aber darauf hin, dass sich hinter den in der dreidimensionalen Welt dargestellten Objekten die Anwendungsdaten verbergen, welche nach dem MVC-Muster von einer Modellkomponente verwaltet werden sollen. Die Modellkomponente ist somit zu dieser Einteilung hinzuzufügen. Sowohl die Manipulations- als auch die Zugriffskomponente steuern das Verändern von Daten der dreidimensionalen Welt, wobei die Manipulationskomponente die Darstellung vorhandener Daten in der dreidimensionalen Welt ändert, währenddessen die Zugriffskomponente neue Daten einfügt oder entfernt. Die Daten in der dreidimensionalen Welt sind Visualisierungen der eigentlichen Anwendungsdaten, welche in der Modellkomponente verwaltet werden. Daher werden Änderungen durch die Manipulations- oder Zugriffskomponente in der dreidimensionalen Welt als Anfragen an die Modellkomponente weitergereicht und verarbeitet. Die vorgestellte konzeptionelle Einteilung nach (Barrilleaux, 2001) verfeinert somit das MVC-Muster in den beiden Komponenten Sicht und Steuerung und lässt die Modellkomponente unverändert.

13.2.3 Die einzelnen Komponenten dreidimensionaler grafischer Benutzungsschnittstellen

Zur Entwicklung einer dreidimensionalen grafischen Benutzungsschnittstelle ist es jetzt sinnvoll, die oben vorgestellten funktionalen Komponenten dreidimensionaler grafischer Benutzungsschnittstellen näher zu betrachten.

Steuerung (Control)

Die Steuerungskomponente interpretiert Benutzereingaben als sinnvolle Aktionen in einer Anwendung. Das Hauptproblem liegt dabei in der Zuordnung von zweidimensionalen Eingabe-

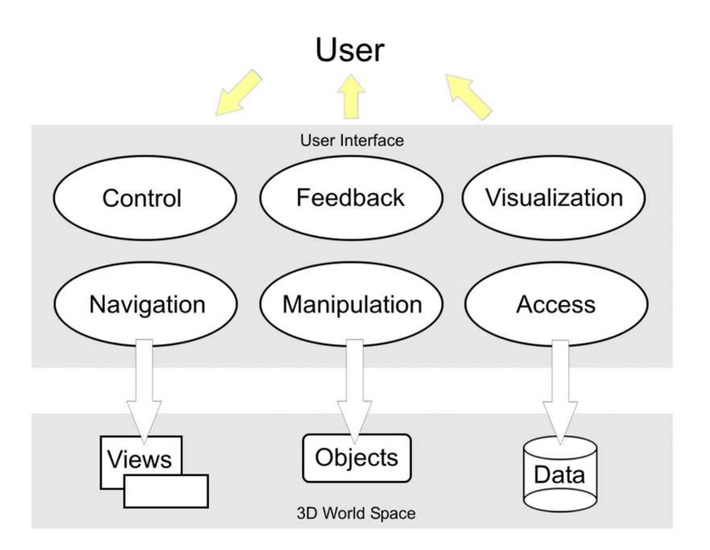


Abbildung 13.2.: Funktionale Komponenten dreidimensionaler Benutzungsschnittstellen nach (Barrilleaux, 2001)

koordinaten zu dreidimensionalen Zielkoordinaten, da nur die Maus als primäres und eben zweidimensionales Eingabegerät zur Verfügung steht. Gelöst wird dieses Problem nach (Barrilleaux, 2001) durch das Coordinate Mapping-Verfahren, welches zum einen festlegt, welche Dimension der zweidimensionalen Eingabe mit welcher Dimension des dreidimensionalen Zielobjekts verbunden wird, und zum anderen die Information über die zusätzliche Dimension des Zielobjekts ermittelt. Je nach beabsichtigter Operation, wie etwa Bewegen oder Rotieren eines Objekts, und je nach aktuellem Blickwinkel werden zwei Eingabekoordinatenachsen direkt mit zwei Koordinatenachsen der dreidimensionalen Welt belegt. Die Information über die dritte Dimension wird dabei durch gute Annahmen über die beabsichtigte Operation, durch Einbeziehung zusätzlicher Nutzerinformationen und durch die Verwendung von Feedbackelementen gefunden. Zu beachten ist hierbei, dass Änderungen der Sicht auch Änderungen der Koordinatenzuordnung zur Folge haben, um weiterhin eine intuitive Steuerung zu gewährleisten. Weiterhin muss bei der Steuerung unterschieden werden, ob gerade ein Datenobjekt in der dreidimensionalen Welt oder die Sicht selbst bewegt wird.

Grundlegend hat die Steuerung natürlich die Aufgabe, Mausbewegungen zu deuten. Hierbei sind die bekannte Möglichkeiten wie Anzahl und Art der Klicks und Drag & Drop mit sinnvollen Aktionen zu belegen.

Feedback

Eine Zentrale Rolle in der dreidimensionalen grafischen Benutzungsschnittstelle übernimmt die Feedbackkomponente, denn sie ist die Möglichkeit für den Entwickler, die Nutzer durch die Anwendung zu führen und sie zu informieren. Im Gegensatz zur realen Welt fehlen nämlich Möglichkeiten, Objekte und ihre Beziehungen intuitiv zu erfassen. Feedbackelemente lassen sich nach ihrem Zweck in die zwei Kategorien Information und Steuerung einteilen.

Während Feedbackelemente zur Information die Datenobjekte und Objektbeziehungen beschreiben, geben Feedbackelemente zur Steuerung konkrete Handlungshinweise. Um diese Aufgaben zu erfüllen, können laut (Barrilleaux, 2001) die im Folgenden vorgestellten Grundelemente verwendet werden. Bezeichner (Identifiers) wie Labels oder Icons gehören direkt zum Erscheinungsbild des zugehörigen Objekts in der dreidimensionalen Welt und benennen dies. Hingegen sind Ausrufe (Callouts) Beschreibungen, die nicht mit der Darstellung des Objekts verbunden sind. Sie lassen sich frei neben ihrem Bezugsobjekt platzieren und sind meist durch eine Linie mit ihm verbunden. Von modernen grafischen Benutzungsschnittstellen bekannt sind Arbeitshinweise (Tooltips), deren großer Vorteil im Gegensatz zu Ausrufen darin besteht, dynamisch erscheinen zu können. Zur Steuerung dienen Indikatoren (Indicators) und Griffe (Handles), die suggestiv geformt sind, um dem Nutzer einen Handlungshinweis, wie z. B. Rotieren eines ausgewählten Objekts, zu geben. Zum Anzeigen der ausgewählten Aktion dient die Mauszeigerform, welche in vielfältiger Weise geändert werden kann. Des Weiteren können auch durch Soundeffekte Handlungshinweise gegeben werden. An das Design und die Visualisierung von Feedbackelementen werden einige Anforderungen gestellt, damit sie ihre Aufgabe gut erfüllen können. Sie müssen sich leicht von den Objekten in der dreidimensionalen Welt unterscheiden lassen, gut lesbar sein, aber trotzdem möglichst wenig von der Welt überdecken. Wird ihr Bezugsobjekt bewegt, sollen sie weiterhin nutzbar bleiben und sich dabei möglichst von selbst neu anordnen, so dass diese Aufgabe dem Nutzer erspart bleibt. Zu viele Feedbackelemente können unübersichtlich wirken, deshalb ist dynamisches Erscheinen sinnvoll.

Visualisierung (Visualization)

Visualisierung ist der Prozess, durch den eine Anwendung ihre Daten dem Nutzer präsentiert. Techniken zur Visualisierung sind in großer Zahl vorhanden, die beste Form für eine Anwendung hängt von ihren Daten und Nutzern ab. Da mit unserem Editor Klassendiagramme erstellt werden sollen, sind wir im Bereich der abstrakten Daten, die kein Analogon in der realen Welt haben. Für das Design der virtuellen dreidimensionalen Welt stehen daher viele Möglichkeiten offen, wobei stets das Ziel angestrebt werden sollte, dem Nutzer durch die Darstellung ein besseres Verständnis für die Daten zu geben. Bei der Visualisierung abstrakter Daten ist es wichtig, eine künstliche Orientierung einzuführen, damit der Nutzer wie gewohnt durch die dreidimensionale Welt navigieren kann.

In dreidimensionalen Welten tritt ständig das Problem auf, dass Objekte überdeckt gezeichnet werden müssen. Hierzu existieren nach (Barrilleaux, 2001) unter anderem die Lösungen X-Ray overlay und Line of sight. Bei der X-Ray overlay-Technik besitzt ein Objekt zwei Formen, eine für normale Sichtbarkeit und eine für Überdeckung. Beispiele für Überdeckungsformen sind Semitransparenz, gestrichelte Linien oder Helligkeit. Eine intelligente Lösung des Problems stellt die Line of sight-Technik dar. Die Anwendung entscheidet aktiv, ob Objekte in der Sichtlinie des Nutzers zu seinen Beobachtungsobjekten liegen und macht diese automatisch unsichtbar. Eine weitere für den Nutzer angenehme Visualisierungstechnik sind mehrfache Sichten (Multiple views). Durch den Einsatz von mehreren Darstellungen, wie z. B. einer Überblickskarte und einen Detailausschnitt, behält der Nutzer leichter die Orientierung.

Navigation

Für die Veränderung des Nutzerstandortes in der virtuellen Welt sorgt die Navigationskomponente. Man unterscheidet dabei laut (Barrilleaux, 2001) zwei Navigationsformen: Räumliche — und kontextuelle Navigation. Die räumliche Navigation bezieht sich auf die Steuerung von Sichtobjekten, wobei man generell drei Möglichkeiten hat. Aus der Sichtweise einer ersten Person ist der Nutzer selbst das Sichtobjekt und kontrolliert es aus dieser Perspektive. Autofahren ist hierfür ein sehr gutes Beispiel. Navigation aus der Sicht einer zweiten Person benötigt Interaktion mit Objekten und Steuerungselementen in der dreidimensionalen Welt. Dies wird am einfachsten anhand des Beispiels der objektzentrierten Sicht klar: Der Nutzer wählt ein Beobachtungsobjekt aus und bewegt anschließend seinen Blick um dieses Objekt herum, damit er es untersuchen kann. Auch vom Standpunkt einer dritten Person aus lässt sich navigieren. Dafür ist eine separate Steuerung außerhalb der Welt wie z. B. eine Überblickskarte nötig. Ein anderes Konzept verfolgt die kontextuelle Navigation, welche dem Nutzer die Möglichkeit gibt, alternative Darstellungen derselben oder verwandter Daten zu sehen. Dies wird zum Beispiel durch Änderungen des Detailniveaus von Paketen über Klassen hin zu Attributen erreicht.

Für den Entwurf der Navigation ist es sinnvoll, mögliche Beschränkungen einzuführen. So kann die Navigation um einen festen Punkt dem Nutzer helfen, den Überblick zu behalten. Falls der Nutzer die Orientierung in der dreidimensionalen Welt verlieren sollte, ist eine vorher definierte Grundposition, zu der man einfach zurückkehren kann, sinnvoll. Für Präsentationen eignet sich das Mittel geführter Touren durch die dreidimensionale Welt, etwa zu den wichtigsten Klassen. Somit ist es einfacher, sich einen ersten Überblick zu verschaffen.

Manipulation

Die Art und Weise, wie Nutzer mit den Datenobjekten in der dreidimensionalen Welt der Anwendung interagieren, heißt Manipulation. Wie bei der Navigation werden auch hier wieder die drei verschiedenen Kontrollpersonen unterschieden. Zur Objektmanipulation stehen standardmäßig die bekannten Möglichkeiten Selektieren, Skalieren, Verschieben, Rotieren und Färben zur Auswahl. Zusätzlich gibt es die wichtige Möglichkeit, logische Verbindung zwischen Elementen erstellen zu können. Allerdings sollten dem Nutzer nicht in jedem Fall alle Objektmanipulationen zu Verfügung stehen, sondern vielmehr nur die, die im Einklang mit den Möglichkeiten des Anwendungsbereichs liegen. Hier besteht ein Ansatzpunkt zur Umsetzung einer Syntaxsteuerung. Bei der Durchführung von Objektmanipulationen wird der Nutzer durch relationale Feedbackelemente unterstützt. Zur Hervorhebung von selektierten Objekten eignet sich die Markierung ihrer Konturen (Outlines). Wenn der Abstand eines Objekts zum Boden angezeigt werden soll, wird von den Objektumrissen aus, ähnlich eines langen Rocks, das Lot zum Boden gefällt (Skirts). Durch Verbindungspunkte wird angezeigt, ob und wo Objekte miteinander kombiniert werden können (Snaps). Um nun auch die internen Eigenschaften eines Datenobjekts wie z. B. Methoden von Klassen, zu ändern, gibt es Spezifikationsmöglichkeiten. Sie können unter anderem durch gesonderte Dialoge oder objektspezifische Menüs realisiert werden.

Zugriff (Access)

Das Bearbeiten von Daten außerhalb der dreidimensionalen Welt und das Hinzufügen und Entfernen von Daten zur dreidimensionalen Welt wird in der Zugriffskomponente zusammengefasst. Die Präsentation der Daten kann dabei durch Listen, Tabellen, Bäume oder Graphen erfolgen, je nach ihrer Art. Diese weiteren Darstellungsmöglichkeiten können die Verwaltung der dreidimensionalen Daten erleichtern. So kann z. B. die Suche nach Objekten in geordneten Listen einfacher sein, als die ganze dreidimensionale Welt zu erkunden. Auch das Verschieben von hierarchischen Daten in Bäumen lässt sich eventuell schneller durchführen. Für Datenobjekte, welche häufig in die dreidimensionale Welt eingefügt werden, bietet sich eine Datenpalette an, in der die Objekte für den schnellen Zugriff als Icons dargestellt sind. Eine weitere Möglichkeit, Objekte in die dreidimensionale Welt einzufügen, ist die Verwendung einer Vorschau. So kann das Objekt erst in der Vorschau bearbeitet und anschließend eingefügt werden. Da der Bereich der Zugriffskomponente sich mit Datenbearbeitung außerhalb der dreidimensionalen Welt befasst, sind hier wieder Aspekte zweidimensionaler grafischer Benutzungsschnittstellen zu berücksichtigen.

13.3 Fazit

Dieser erste Überblick über das Thema grafischer Editoren und dreidimensionaler grafischer Benutzungsschnittstellen liefert einige Anhaltspunkte für das zu entwickelnde Editorframework. Die Art der Benutzerinteraktion (frei, strukturorientiert, syntaxgesteuert) ist zu bedenken. Für die Gestaltung der Benutzeroberfläche sind Anforderungen wie gute, intuitive Benutzerführung und Softwareergonomie sowie die Einbettung in die zweidimensionale grafische Benutzungsschnittstelle von Eclipse zu beachten. Die durch das MVC-Muster gezeigte Trennung von Model und Darstellung ermöglicht es, austauschbare Sichten auf die Daten zu realisieren. Bei der Erstellung von dreidimensionalen grafischen Benutzungsschnittstellen sind die sechs funktionalen Komponenten Steuerung, Feedback, Visualisierung, Navigation, Manipulation und Zugriff zu entwerfen. Dabei ergeben sich im Bereich der Steuerung, Navigation und Manipulation Fragen zur Umrechnung der zweidimensionalen Mauseingaben, zur Sichtweise (erste, zweite und dritte Person) und zur Beschränkung der Steuerung. Im Bereich der Visualisierung ist die Darstellung abstrakter Daten und das Problem der Überdeckung anzugehen. Sehr wichtig für das Ziel einer intuitiven grafischen Benutzungsschnittstelle ist die vernünftige Verwendung von Feedbackelementen.

TEIL 3

Releasebeschreibungen

KAPITEL 14

Beschreibung des ersten Release

14.1 Einleitung

Kai Gutberlet

Das Ziel des ersten Release ist es, einen einfachen Diagrammtypen zu erstellen, der statische Java-Klassenstrukturen dreidimensional darstellt. Der neue Diagrammtyp soll dabei die im Folgenden beschriebene Syntax haben. Klassen sollen als Würfel dargestellt werden. Die Zugehörigkeit von Klassen zu ihren Paketen soll durch die Farbe der zugehörigen Würfel eindeutig erkennbar sein: Alle Klassenwürfel eines Pakets sollen die gleiche Farbe besitzen. Des Weiteren soll die Zugehörigkeit von Klassen zu einem Paket durch eine gehäufte Anordnung der Klassenwürfel im Raum dargestellt werden. Als Bedingung für die Anordnung der Klassenwürfel im Raum gibt es zusätzlich die Regel, dass erbende Klassen unterhalb der vererbenden Klassen im Raum angeordnet werden sollen.

Dieses Release dient dazu, die Entwickler mit den zu verwendenden Technologien, wie der Eclipse-Plugin-Struktur und der Java-3D-API, vertraut zu machen. Die Anforderungen an das erste Release und deren Umsetzung sind im Folgenden näher beschrieben.

14.2 User Stories

Semih Sevinç

Das Hauptziel des ersten Releases ist es, ein *Eclipse*-Plugin zur dreidimensionalen Visualisierung von Java-Klassen zu erstellen. Als Anforderungsdefinition hierfür dienen die von den Kunden aufgestellten User Stories. Diese unterteilen sich in zwei Bereiche, nämlich einmal in die von den Entwicklern akzeptierten und einmal in die abgelehnten bzw. geänderten User Stories. Im Folgenden werden zunächst die akzeptierten User Stories vorgestellt, beginnend mit den wichtigsten bis hin zu optionalen Anforderungen. Danach werden die abgelehnten bzw. modifizierten User Stories aufgelistet.

14.2.1 Akzeptierte User Stories

Die wichtigste Anforderung ist die dreidimensionale Visualisierung der Klassen eines Pakets. Dafür soll im Package Explorer von *Eclipse* die Möglichkeit gegeben sein, die Klassen eines ausgewählten Pakets in einem neuen externen Fenster dreidimensional darzustellen. Bei erneuter Generierung der 3D-Szene muss beim selben Paket das gleiche Bild geliefert werden.

Zur Darstellung der Klassen ist vorgegeben, dass diese als gleich große Würfel dargestellt werden sollen und bei der 3D-Szenengenerierung sinnvoll und nicht verschiebbar platziert werden. Des Weiteren ist für die Klassen vorgesehen, dass Kindklassen immer auf einer graphisch tieferen Ebene als ihre Vaterklassen angeordnet werden. Dies soll auch über Paketgrenzen hinaus erfüllt sein. Bei der Darstellung der Klassen als Würfel soll durch ihre gehäufte Anordnung im Raum die Zuordnung der einzelnen Klassen zu einem Paket sofort ersichtlich sein. Jede sinnvolle interne Information, beispielsweise die Kameraposition oder der Name der Vaterklasse, soll über ein geeignetes Schnittstelle abfragbar sein. Eine weitere Anforderung der Kunden ist ein Informationsfenster, welches die Statusinformationen zur 3D-Ansicht beinhalten soll, wie z.B. die Koordinaten der Sicht, Anzahl der eingelesenen Klassen oder die Anzahl der gezeichneten Klassen. Weiterhin soll es möglich sein, in der 3D-Szene zu navigieren, wobei eine einfache Navigation mit der Tastatur ausreicht. Die Zuordnung der einzelnen Klassen zu einem Paket soll durch ihre einheitliche Farbgebung ersichtlich sein. Zusätzlich ist erwünscht, dass die Klassen geeignet beschriftet werden. Dabei würde eine Beschriftung der Klassen durch eine Nummerierung ausreichen. Die letzte Anforderung ist, dass es eine Möglichkeit geben soll, die Sicht des Benutzers auf die 3D-Szene auf einen vorher definierten Startpunkt zurück zu setzen. Dabei soll der Startpunkt sinnvoll gewählt sein, so dass man alle Klassen am Anfang überblicken kann.

14.2.2 Abgelehnte User Stories

Es war zunächst von den Kunden gewünscht, dass die Visualisierung eines ausgewählten Pakets entweder in einem neuen Fenster dargestellt wird oder in einem neuen View in *Eclipse*. Die zweite Variante wurde von den Entwicklern für das erste Release abgelehnt. Der Grund hierfür war, dass in der damaligen *Eclipse* Version, *Eclipse* 2.1.1, AWT-Komponenten nicht in SWT-Komponenten eingebunden werden konnten. Außerdem sollte vor der 3D-Szenengenerierung die Möglichkeit gegeben sein, eine Farbe für ein Paket auszuwählen, wobei jedes Paket eine unterschiedliche Farbe haben soll. Die Farbauswahl wurde automatisiert, so dass keine manuelle Auswahl der Farben mehr nötig ist. Des Weiteren wurde die Anforderung, dass Klassen von einer transparenten Pakethülle umgeben sind, aus zeitlichen Gründen von den Entwicklern abgelehnt.

14.3 Systemmetapher

Daniel Unger, Jan Wessling

Die Systemmetapher ist die geplante Architektur, welche als Grundlage für die Vorgehensweise bei der Implementierung in dem Release dient. Dabei stützt sich das Release auf das Model-View-Controller-Konzept (MVC). Dieses Konzept unterteilt sich in drei Bereiche. Das Model repräsentiert die Daten einer Anwendung, kann deren Zustand liefern und Veränderungen an ihnen vornehmen. Der View dient der graphischen Darstellung der Daten und somit einer visuellen Darstellung des Models. Der Controller definiert, wie der View Zugriff auf das entsprechende Modell hat. Im den folgenden Abschnitten wird gezeigt, wie diese einzelnen Bereiche in unserem Projekt umgesetzt worden sind. Einen Überblick über die so entstandene

Architektur bietet das Klassendiagramm in Abbildung 14.2

14.3.1 Model

Im Modell werden die Informationen zu den Daten gehalten. Für das Release sind dabei die Informationen wichtig, die zur Darstellung von Objekten im dreidimensionalen Raum benötigt werden. Dabei beruht das in dem ersten Release verwendete Datenmodell auf dem von der benutzten Entwicklungsumgebung *Eclipse* bereitgestellten Datenmodell und wird mit der Klasse `EclipseDatenmodell` realisiert. Allerdings erweitert es dieses noch um für das Release notwendige Punkte. Dieses sind Informationen, die für die entsprechende Anordnung eines Objektes im dreidimensionalen Raum benötigt werden. Die Anordnung ist abhängig von der Paketzugehörigkeit sowie der Vererbungsbeziehung einer Klasse. Weiterhin wird die berechnete, logische Ausrichtung abgespeichert. Das Speichern geschieht dabei in Klassen- bzw. Paket-Containern, die an das bereitgestellte Modell angehangen werden.

14.3.2 View

Die zum View gehörenden Klassen dienen zur Darstellung der im Modell gehaltenen Daten. In dem Release sollen Klassen ihrer Paketzugehörigkeit und ihrer Vererbungsbeziehungen entsprechend in 3D dargestellt werden. Dieser dynamische Vorgang wird in Abbildung 14.1 gezeigt. Es wird eine Szene von dem Modell erzeugt, welche in einem Fenster dargestellt wird. Dazu wird mittels den zum Controller gehörenden Klassen `MenuePunkt` und `UmrechnungIn3DModell` eine Instanz der View-Klasse `Szene3DModell` erzeugt. Weiterhin läßt sich mit der zur View gehörenden Klasse `Navigator` durch diese Szene navigieren. Dabei fragt diese von der Klasse `Editor` den Startpunkt ab und kann anschließend durch das von der View-Klasse `Darstellungsfenster` erzeugte Fenster navigieren. Das Fenster wird mittels des `Editor` geöffnet und stellt das `Szene3DModell` dar. Ebenso gibt es noch ein Fenster, in dem wichtige Informationen zu dem Modell angezeigt werden, den `OutlineView`. Die dazugehörige View-Klasse ist `EclipseOutline`. Der `OutlineView` wird ebenfalls durch den `Editor` gestartet.

14.3.3 Controller

Die Klassen, die als Controller dienen, sind primär für die Interaktion des Views mit dem Model zuständig. Indirekt wird dadurch auch definiert, wie der Benutzers mit der Applikation interagieren kann. Weiterhin sollen die Controllerklassen die im Modell gehaltenen Daten bearbeiten können. Der dabei eingehaltene Ablauf ist in Abbildung 14.3 dargestellt. In diesem Release wird mittels des Controllers eine Berechnung ausgeführt, um die Daten (Klassen und Pakete) später in dem 3D-Modell richtig anzuordnen. Weiterhin schreiben sie diese neu gewonnen Informationen in das bereits vorhandene Datenmodell. Der dynamische Ablauf sieht dabei so aus, dass über einen Menüpunkt zunächst der Editor sowie die Umrechnung in das 3D-Modell gestartet wird. Diese Umrechnung stößt selber nun die einzelnen Berechnungen an. Diese unterteilt sich in drei Teile. Es gibt eine Berechnung der Farben, bei der jede Klasse gemäß ihrer Paketzugehörigkeit eine spezielle Farbe erhält. Alle Klassen eines Paketes werden in der gleichen Farbe dargestellt. Die Analyse der Ebeneninformationen ordnet die

Klassen auf verschiedene Ebenen und somit vertikal an. Dabei geschieht die Anordnung aufgrund der Vererbungs-Hierarchien der Klassen. Klassen, die von keiner anderen Klasse erben, stehen auf der höchsten Ebene. Anschließend wird absteigend sortiert, so dass erbende Klassen immer mindestens eine Ebene tiefer als ihre vererbende Klasse, aber niemals über dieser liegen. Durch die Analyse der Haufen werden die Klassen entsprechend ihrer Paketzugehörigkeit auf der X-Z-Ebene angeordnet. Klassen, welche zu einem Paket gehören sind somit räumlich zueinander angeordnet. Die durch die Berechnung erhaltenen Daten werden in das erweiterte *Eclipse*-Datenmodell abgespeichert. Aufgrund dieser Informationen lässt sich anschließend ein geeigneter Startpunkt für die 3D-Darstellung berechnen. Dieser soll so gewählt werden, dass zunächst alle dargestellten Klassen in dem Fenster zu sehen sind. Dabei greift die Umrechnung in das 3D-Modell auf das erweiterte *Eclipse*-Datenmodell zu, wo die nötigen Informationen in den entsprechenden Klassen- und Paket-Containern gespeichert sind. Anschließend berechnet die Umrechnung in das 3D-Modell einen Startpunkt und übermittelt ihn dem Editor.

14.4 Reflexion über die Tasks

Armin Bruckhoff, Christian Mocek, Sven Wenzel

Im Nachfolgenden wird über die Tasks des ersten Release reflektiert. Hierzu werden die einzelnen Tasks kurz vorgestellt und insbesondere auf die signifikanten Probleme und die Differenzen in den Zeitabschätzungen eingegangen.

Die gegebenen User Stories wurden zunächst wie folgt als zwölf explizite Anforderungen aufgeschlüsselt und in einzelne Tasks unterteilt. Für jeden der Tasks wurde anschließend die Zeit abgeschätzt, die zur Umsetzung als nötig empfunden wurde.

14.4.1 Gesamtrahmen für ein Plugin

Plugin-Rahmen für *Eclipse*-Plugin

Beschreibung: Ziel dieses Tasks war es, den eigentlichen Plugin-Rahmen zu implementieren, der dem gesamten Plugin zu Grunde liegt und die Verbindung zum Datenmodell von *Eclipse* bereitstellt.

geplante Zeit: Aufgrund der geringen Erfahrungen mit *Eclipse* haben wir eine Dauer von 8 Stunden veranschlagt.

reale Zeit: Es wurden nur fünf Stunden für die Realisierung benötigt. Der eigentliche Plugin-Rahmen war nicht so komplex wie erwartet. Dafür musste mehr Zeit in den Task „Menüpunkt“ investiert werden.

Menüpunkt „Szene generieren“ für Package Explorer

Beschreibung: In diesem Task wurde der eigentliche Menüpunkt für das Auslösen der Szenengenerierung implementiert. Hierzu war es nötig, den entsprechenden *Extension Point* von *Eclipse* zu verwenden.

geplante Zeit: Zwei Stunden. Es waren im Wesentlichen nur die entsprechenden *Extension Points* zu ermitteln und die Frage zu klären, wie eingestellt werden kann, dass der Menüpunkt nur auf bestimmte Einträge im *Package Explorer* reagiert.

reale Zeit: Vier Stunden. Aufgrund der mangelnden Erfahrungen war die Realisierung des Menüpunkts umfangreicher als erwartet.

Datenmodell realisieren

Beschreibung: Das Datenmodell von *Eclipse* musste für unsere Bedürfnisse erweitert werden. Alternativ stand die Implementierung eines eigenen Modells zur Wahl.

geplante Zeit: 14 Stunden. Es war ursprünglich geplant, das Datenmodell komplett selbst zu implementieren, d.h. es hätte auch ein entsprechender Parser für die Java Dateien geschrieben werden müssen.

reale Zeit: Zehn Stunden. Das Datenmodell von *Eclipse* war gut dokumentiert und ließ sich über Ressourcen erweitern, so dass es nicht nötig war, eine eigene Datenstruktur zu entwickeln.

14.4.2 Darstellung der Klassen

Darstellen eines Würfels in Java3D

Beschreibung: Erzeugen eines Java3D-Würfelobjekts.

geplante Zeit: Eine Stunde. Es mussten nur die entsprechenden Koordinaten in einem Objekt gespeichert werden, aus dem dann ein Shape3D erzeugt wird.

reale Zeit: Sechs Stunden. Die Einarbeitung in Java3D war zeitaufwendiger als erwartet, da Java3D viele komplexe Möglichkeiten bereitstellt, 3D-Objekte zu visualisieren. Des Weiteren mußte für den Level-of-detail (LOD) die Klasse des Würfels erheblich erweitert werden.

14.4.3 Viewer

Szenegraph generieren

Beschreibung: Um die Würfel im Raum geordnet anzuzeigen, muss für Java3D eine Szene generiert werden.

geplante Zeit: Acht Stunden. Es lag zur Zeitabschätzung noch keine Erfahrung mit Java3D vor, von daher wurde die Zeit geschätzt.

reale Zeit: 15 Stunden. Wie im Task für den Würfel wurde auch hier die Einarbeitung in Java3D unterschätzt.

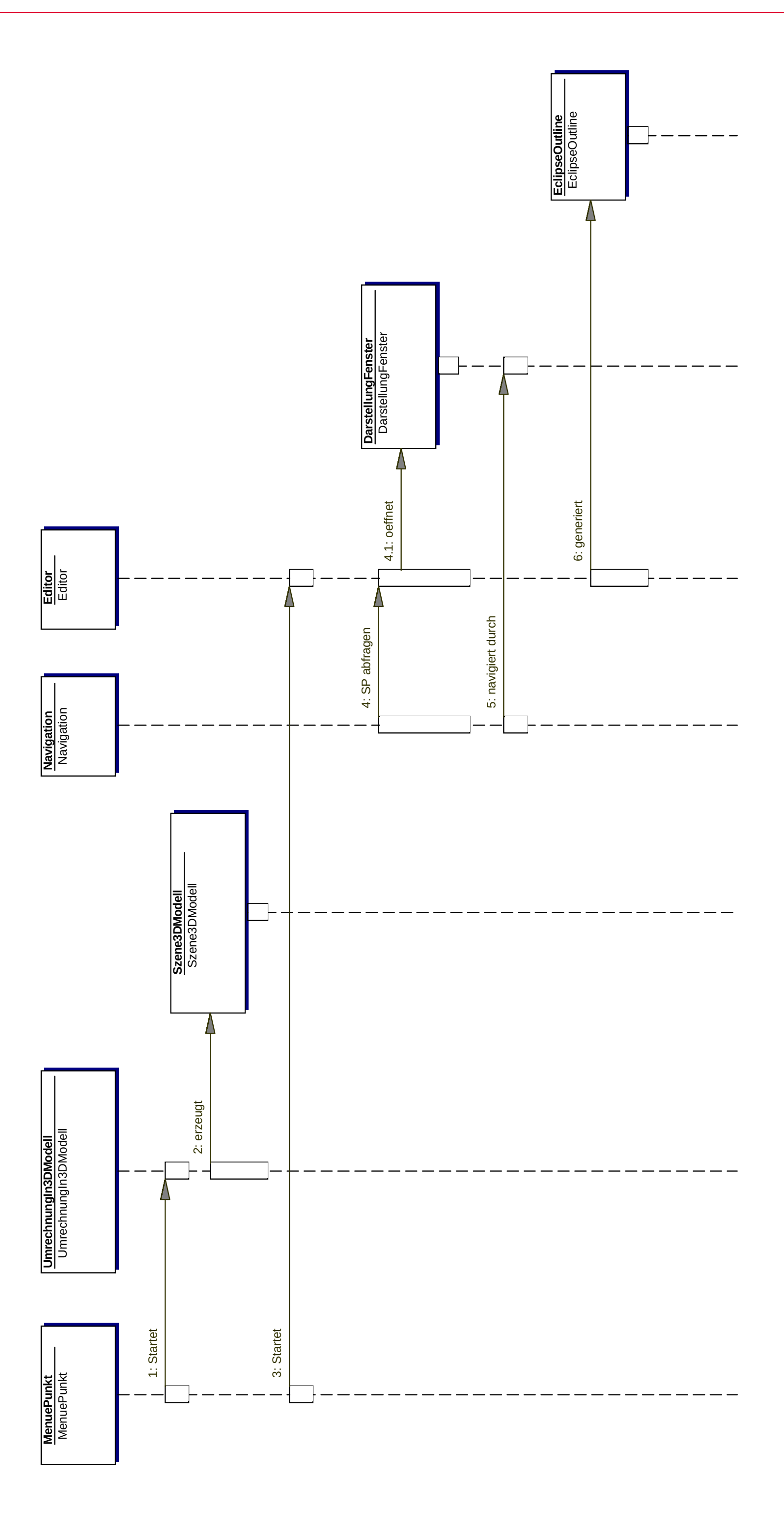


Abbildung 14.1.: Sequenzdiagramm zur Erzeugung der Darstellung

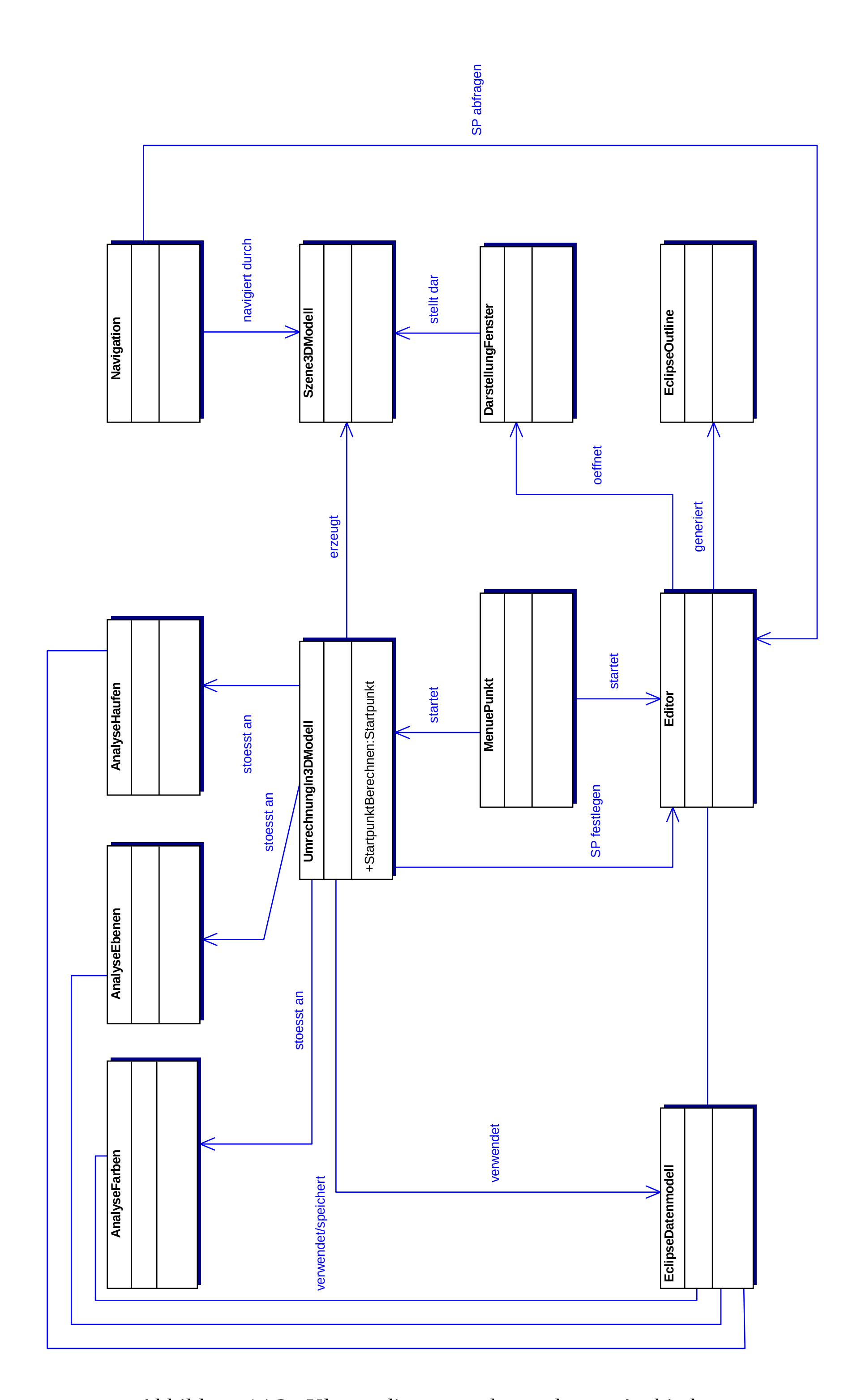


Abbildung 14.2.: Klassendiagramm der geplanten Architektur

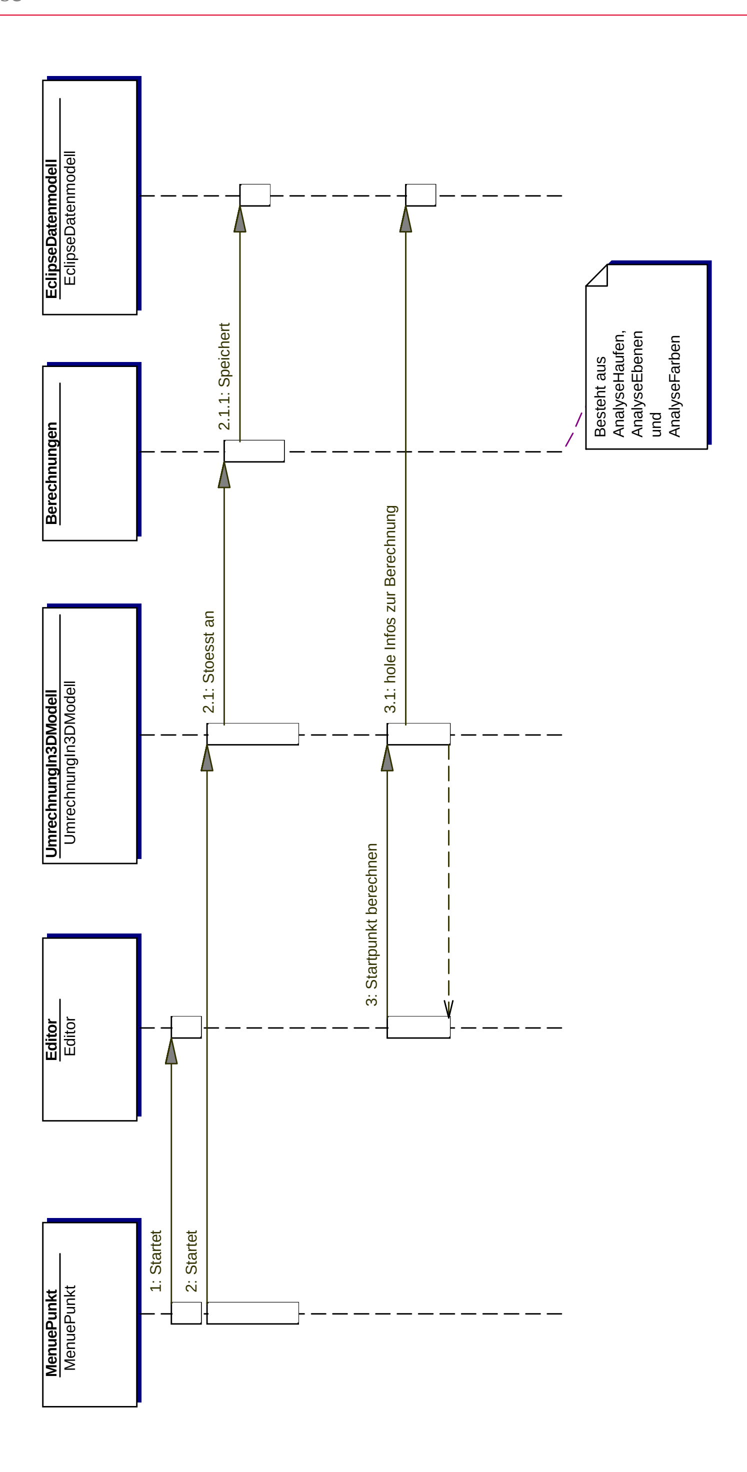


Abbildung 14.3.: Sequenzdiagramm des Berechnungsvorgangs

Öffnen eines Fensters für 3D-View

Beschreibung: Da die Anzeige von Java3D-Komponenten innerhalb von *Eclipse* nicht möglich ist, wurde ein eigenes Fenster zur Anzeige dieser Komponenten benötigt.

geplante Zeit: Zwei Stunden wurden eingeplant.

reale Zeit: Die geplante Zeit konnte eingehalten werden.

Generierung des Universums

Beschreibung: Um die 3D-Szene sinnvoll für unsere Anwendung zu konfigurieren, sollte ein eigenes Universum realisiert werden.

geplante Zeit: Vier Stunden. Es war geplant einfach einen blauen Hintergrund und eine Ebene im Raum zu zeichnen, welche dann als Basis für das Universum gelten sollten.

reale Zeit: Acht Stunden. Wie beim Würfel wurde auch hier die Einarbeitung in Java3D unterschätzt. Es mussten Algorithmen entwickelt werden, die die Größe der Ebene an die maximale Ausdehnung der Paketdarstellung anpassen. Die Größe des Universums musste dynamisch angepasst werden, weil sonst die Darstellung von Java3D zu langsam geworden wäre.

14.4.4 Ebenen

Berechnung der logischen Ebenen

Beschreibung: Um die Klassen gemäß der Vererbungshierarchie anzuordnen, sollten sie verschiedenen logischen Ebenen zugeordnet werden.

geplante Zeit: Vier Stunden. Erste Ideen für einen entsprechenden Algorithmus deuteten darauf hin, dass dessen Umsetzung ca. vier Stunden dauert.

reale Zeit: Drei Stunden.

14.4.5 Häufung

Anordnen der Klassen im Raum

Beschreibung: Die Klassen eines Pakets auf einer Ebene sollten reproduzierbar angeordnet werden.

geplante Zeit: 16 Stunden. Es waren zunächst keinerlei Ideen vorhanden, wie man so etwas effizient realisieren könnte.

reale Zeit: Zehn Stunden. Es wurde ein sinnvoller Algorithmus zur Verteilung und Berechnung der Positionen gefunden, der sich zudem einfach implementieren ließ.

Information, wie viele Klassen ein Paket auf einer Ebene besitzt

Beschreibung:	Diese Information wurde für die „Anordnung im Raum“ benötigt.
geplante Zeit:	Vier Stunden.
reale Zeit:	Vier Stunden.

Einteilen des Raums in Bereiche (ohne Höheninformation) für die einzelnen Pakete

Beschreibung:	Die Klassen eines Pakets sollten räumlich nah beieinander angeordnet sein.
geplante Zeit:	Zwei Stunden.
reale Zeit:	Drei Stunden. Aufgrund der gleichzeitigen Erledigung des Tasks „Optimieren der Anordnung“ wurde hier mehr Zeit benötigt.

Optimieren der Anordnung

Beschreibung:	Die Anordnung der Klassen soll sinnvoll erfolgen. Abstände sollten weder zu groß noch zu klein sein.
geplante Zeit:	Vier Stunden.
reale Zeit:	Keine. Ist mit dem Task „Einteilen des Raums“ erledigt worden.

14.4.6 Abfragen

Bereitstellen von get-Methoden

Beschreibung:	Die verschiedenen Klassen des Projektes sollten *get()*-Methoden bereitstellen, um die Tests sinnvoll realisieren zu können.
geplante Zeit:	Keine, da diese Aufgabe in jedem anderen Task enthalten war und eher als Konvention für die Entwicklung galt.
reale Zeit:	Keine.

14.4.7 Infofenster

Abfragen wichtiger Informationen

Beschreibung:	Es sollten sinnvolle Informationen abgefragt werden, um sie im Infofenster anzuzeigen.
geplante Zeit:	Vier Stunden waren eingeplant.
reale Zeit:	Keine, da im Task „Gestalten eines Infofenster“ enthalten.

Listenerfunktionalität

Beschreibung: Das Infofenster sollte als Listener auf Änderungen in *Eclipse* reagieren.
geplante Zeit: Eine Stunde wurde veranschlagt.
reale Zeit: Eine Stunde.

Gestalten eines Infofensters
Beschreibung: Es galt ein Infofenster zu entwerfen, dass die sinnvollen abgefragten Informationen anzeigen kann.
geplante Zeit: Fünf Stunden. Es war geplant den *Outline-View* zu verwenden.
reale Zeit: Acht Stunden. Da der *Outline-View* nur mit Editoren zusammenarbeitet, musste das Verhalten in einem eigenen View nachgebaut werden.

14.4.8 Navigation

KeyNavigatorBehavior einbinden
Beschreibung: Das *KeyNavigatorBehavior* dient zur Navigation mit der Tastatur durch die dargestellte dreidimensionale Szene.
geplante Zeit: Drei Stunden. Das Verhalten von Behaviors unter Java war unbekannt und somit musste man sich dieses Wissen erst anlesen.
reale Zeit: Eine Stunde. Es zeigte sich, dass das Wissen bei einem Entwickler doch aus anderen Projekten bekannt war.

14.4.9 Farbauswahl

Sinnvolle automatische Wahl einer Farbe, abhängig von den bereits vergebenen Farben
Beschreibung: Die Würfel eines Pakets sollten mit der gleichen Farbe gekennzeichnet werden. Zudem sollte die Farbe bei jeder Szenegenerierung für ein Paket unverändert bleiben.
geplante Zeit: Sechs Stunden.
reale Zeit: Sechs Stunden.

14.4.10 Farbgebung

Einfärben eines Würfels
Beschreibung: Die zuvor bestimmte Farbe sollte dem jeweiligen 3D-Würfel zugewiesen werden.

geplante Zeit: Eine Stunde. Farbverwaltung unter Java 3D war unbekannt, jedoch wurde vermutet, dass es reichen würde, die einzelnen Knoten des Würfels einzufärben.
reale Zeit: Zehn Minuten. Das Wissen war bereits aus dem Task der Würfelerstellung vorhanden.

Merken der Farbe für ein Paket
Beschreibung: Da alle Klassen eines Pakets dieselbe Farbe haben sollten, mussten sich die entsprechenden Informationen gemerkt werden.
geplante Zeit: Eine Stunde. Zunächst sollten die Farben manuell vergeben werden können.
reale Zeit: Keine. Die Farben werden nach einem deterministischen Algorithmus vergeben. Somit war dieser Task im Task „Sinnvolle automatische Wahl einer Farbe“ enthalten.

14.4.11 Beschriftung

Erzeugen einer Beschriftung aus dem Klassennamen
Beschreibung: Um die einzelnen Klassen identifizieren zu können, sollten sie beschriftet werden.
geplante Zeit: Zwei Stunden. Die Idee hierbei war, so genannte *Billboards* zu verwenden.
reale Zeit: Eineinhalb Stunden. Ließ sich mit `OrientedShape3D` gut realisieren.

Einfügen der Beschriftung in den Szenegraphen
Beschreibung: Die Beschriftung einer Klasse sollte auch in der Szene angezeigt werden.
geplante Zeit: Zwei Stunden. Die Idee hierbei war, so genannte *Billboards* zu verwenden.
reale Zeit: Eineinhalb Stunden. Ließ sich mit `OrientedShape3D` gut realisieren.

14.4.12 Startpunkt

Merken eines Startpunktes
Beschreibung: Um die Orientierung für den Anwender zu erleichern, sollte er zu dem Startpunkt der Szene navigieren können. Das Merken des Punktes war damit unumgänglich.
geplante Zeit: Eine Stunde.

reale Zeit:	Keine. war in „Kameraposition verändern“ enthalten.

Kameraposition verändern

Beschreibung:	Für die Navigation zum Startpunkt musste die Kameraposition verändert werden können.
geplante Zeit:	Eine Stunde.
reale Zeit:	Zwei Stunden. Der Task „Merken eines Startpunktes“ war hier miteingeflossen.

Sinnvolle Wahl eines Startpunktes

Beschreibung:	Es war ein sinnvoller Startpunkt für eine Szene zu wählen, von dem aus die Szene gut überschaubar ist.
geplante Zeit:	Vier Stunden.
reale Zeit:	Fünf Stunden.

Abfangen der Tastaturkombination

Beschreibung:	Zum Zurückkehren zur Startposition wurde eine Tastenkombination verlangt.
geplante Zeit:	Eine Stunde.
reale Zeit:	Eine halbe Stunde. Im `KeyNavigatorBehavior` war diese Funktionalität bereits vorhanden. Es wurden lediglich einige Tastenbelegungen angepaßt.

14.4.13 Fazit

Trotz einiger Abweichungen in den einzelnen Tasks ist die geplante Gesamtdauer von insgesamt 101 Stunden realistisch gewesen. Es wurden insgesamt 97 Stunden benötigt.

Durch die Möglichkeit des Zugriffs auf einige vorhandene Funktionalitäten von *Eclipse*, wie z.B. das Datenmodell, konnte in einigen Tasks Arbeitszeit eingespart werden. Diese gewonnene Zeit musste jedoch in die Java3D-bezogenen Tasks zusätzlich investiert werden, da die Einarbeitung in die API von Java3D deutlich umfangreicher war als erwartet.

14.5 Vorstellung der implementierten Architektur

Stephan Eisermann, Kai Gutberlet, Michael Pflug

Im Nachfolgenden soll die in diesem Release implementierte Architektur vorgestellt werden. Dazu wird zunächst die geplante Architektur beschrieben, dann die realisierte Architektur festgehalten und abschließend beide verglichen.

14.5.1 Beschreibung der geplanten Architektur

Das Modell gliedert sich wie in Abschnitt 14.3 näher beschrieben nach dem Model-View-Controller-Konzept (MVC). Zur Datenhaltung sollte das *Eclipse*-Datenmodell passend durch Containerobjekte erweitert werden. An dieser Stelle ist das Modell nur grob definiert, und muss verfeinert werden. Die Klassen des Controllers haben die Aufgabe, mit den Daten des *Eclipse*-Datenmodells Berechnungen zur 3D-Darstellung durchzuführen. Diese Aufgabe soll die Klasse `UmrechnungIn3DModell` übernehmen. Hierfür ist es nötig, aus den Daten zunächst logische Informationen zu berechnen. Diese logischen Informationen sind eine Voraussetzung für die Erzeugung der 3D-Szene und gliedern sich in die Analyse der Farben, die Analyse der Ebenen und die Analyse des räumlichen Zusammenhangs, welche ebenfalls von eigenen Klassen durchgeführt werden. Die logischen Informationen werden in dem erweiterten *Eclipse*-Datenmodell gespeichert.

14.5.2 Beschreibung der realisierten Architektur

Abbildung 14.4 zeigt die implementierte Architektur, welche sich wie folgt unterteilt: Die Aspekte des Modells (rot dargestellt) übernimmt das Paket `datamodel`. Dieses Paket besteht aus den Klassen `ClassContainer`, `Container` und `DataModelController`. Die Klassen `ClassContainer` und `PackageContainer` erweitern das *Eclipse*-Datenmodell und nehmen unsere spezifischen Informationen auf. Diese Erweiterung wird über das *Eclipse*-interne Interface `IRessource` möglich, dass sowohl von den `PackageFragements` als auch von den `ICompilationUnits` implementiert wird. Dieses Interface erlaubt es, unsere Containerobjekte an die entsprechenden Objekte des Elipse-Datenmodells zu binden, welche einzelne Klassen oder Pakete darstellen. Der Zugriff auf diese Objekte erfolgt ebenfalls über das Interface `IRessource`. Die Klasse `DataModelController` initialisiert die Erweiterung des *Eclipse*-Datenmodells.

Den Controlleraspekt (gelb dargestellt) in unserer Implementierung spiegelt die Klasse `Controller` wieder. Von ihr wird die Berechnung der logischen Informationen, die Berechnung der Darstellung sowie die Visualisierung angestoßen.

Die drei Klassen `ColorCalculation`, `LevelCalculation` und `ClusterCalculation` nutzen als Grundlage für die Berechnungen der logischen Informationen das erweiterte *Eclipse*-Datenmodell. Die von diesen drei Klassen berechneten Informationen werden in dem erweiterten Datenmodell gespeichert.

Die Berechnung der 3D-Darstellung erfolgt durch die Klasse `Scene`. Diese nutzt die im erweiterten Datenmodell abgespeicherten logischen Informationen als Grundlage für die Erzeugung der 3D-Szene. Aus dieser Berechnung resultieren Java3D Komponenten, welche die Visualisierung der logischen Informationen wiederspiegeln.

Die Darstellung (grün dargestellt) unterteilt sich in Java3D- und Swingkomponenten. Die Klasse `FrameOpener` enthält die Swingkomponente JFrame, welche die Java3D-Komponente Canvas3D aufnimmt. Alle anderen Java3D-Komponenten werden an das Canvas3D gebunden. Die Klasse `Universe` enthält die von der Klasse `Scene` berechnete 3D-Darstellung und die Navigation. Zusätzlich schafft die Klasse noch einen Rahmen für die Darstellung, wie z.B. einen Boden und ähnliches.

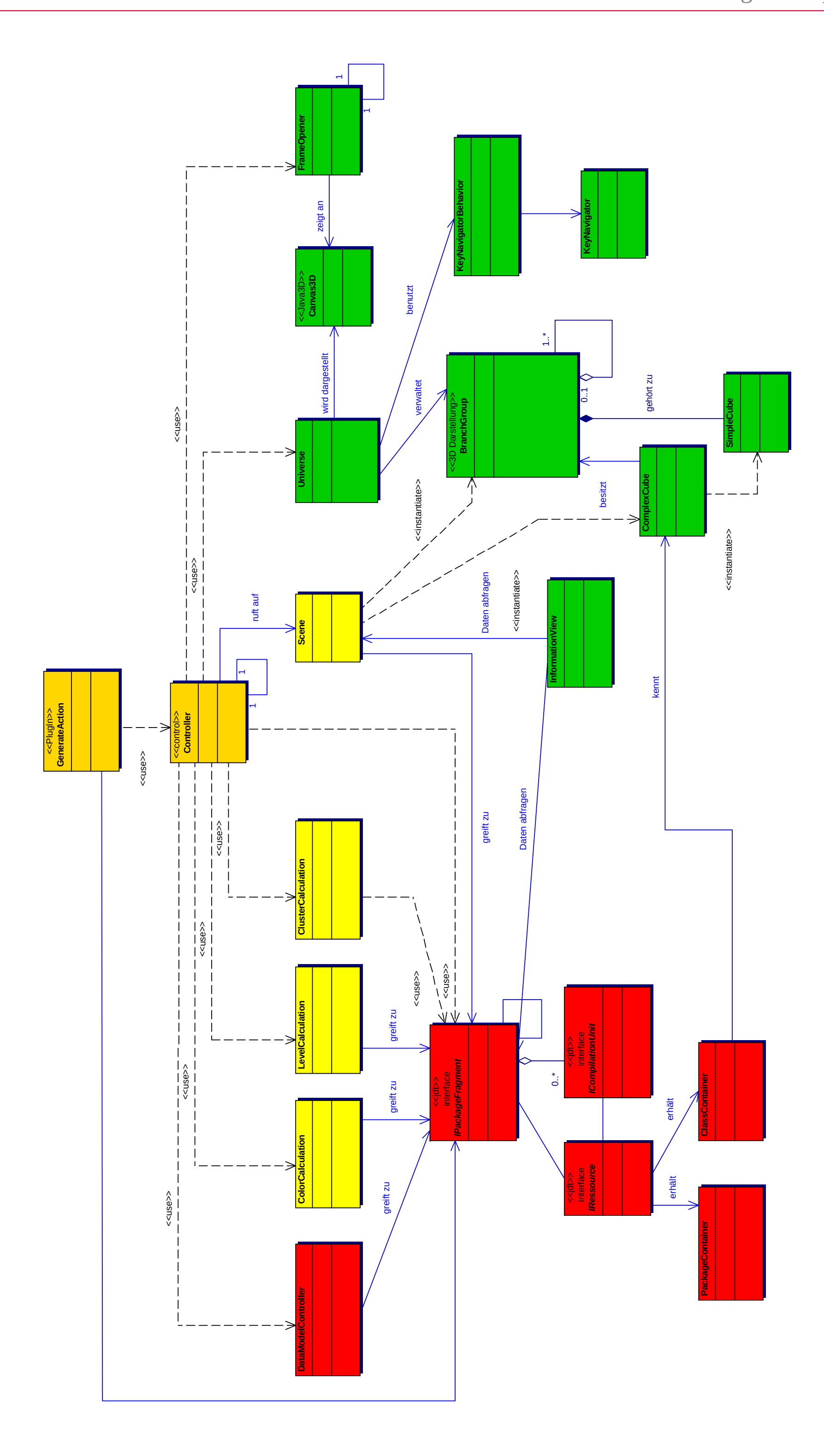


Abbildung 14.4.: Realisierte Systemarchitektur Release 1

14.5.3 Vergleich geplanter- und realsierter Architektur

Zunächst ist zu sehen, dass die geplante MVC-Architektur in der realiserten Architektur umgesetzt worden ist. Die Komponente `Menuepunkt` aus der Systemmetapher entspricht im aktuellen Modell der Klasse `GenerateAction` und bietet die geplante Funktionalität, aus dem *Eclipse*-Pluginkern unser Effects-Plugin aufzurufen. Diese Klasse ist somit der Einstiegspunkt in unser Plugin.

Eine auffällige Änderung in der Steuerung des Programmablaufs ist, dass in der implementierten Architektur nun eine Klasse `Controller` existiert, die die geplanten Komponenten `UmrechnungIn3DModell` und `Editor` nun zusammen repräsentiert. Die Klasse `Controller` übernimmt die zentrale Steuerung aller Aktionen während der Programmausführung und beinhaltet nun keine Berechnung von Informationen mehr.

Weiterhin wurde das *Eclipse*-Datenmodell um eigene Containerklassen erweitert, um zusätzliche Informationen wie Farbe, Ebenen und Anordnung der entsprechenden Klassen und Pakete zu speichern. Die Komponente `Eclipsedatenmodell` wurde in der realisierten Architektur entsprechend des *Eclipse*-internen Datenmodells durch die Interfaces `IPackageFragment`, `IRessource` und `ICompilationUnit` angepasst. Hierbei wird außerdem deutlich, an welcher Stelle das *Eclipse*-interne Datenmodell von uns entsprechend erweitert wurde. Um die Containerklassen an die entsprechenden Objekte zu binden, wurde die Klasse `DataModelController` erzeugt, die somit also die Komponente `EclipseDatenmodell` der geplanten Architektur steuert.

Die Anaylse-Komponenten der geplanten Architektur (Farbe, Cluster, Level) wurden übernommen und durch die drei Klassen `ColorCalculation`, `LevelCalculation` und `ClusterCalculation` realisiert. Die in der ursprünglichen Architektur vorgenommene Trennung der Berechnung der einzelnen logischen Informationen wurde also auch beibehalten.

Im Gegensatz zur Systemmetapher wurde die Klasse `Scene` aus der Komponente `UmrechnungIn3DModell` herausgelöst. Sie hat ausschließlich die Aufgabe, die graphische Darstellung zu berechnen. Somit findet in der Komponente `UmrechnungIn3DModell` keine Berechnung mehr statt. Hier ist die Struktur soweit verbessert worden, dass Steuerung und Berechnung voneinander getrennt wurden.

Umfassende Änderungen der geplanten Architektur wurden insbesondere im Bereich des Views vorgenommen. Dies läßt sich im Wesentlichen darin begründen, dass während der Entwicklung ein weitreichendes Verständnis der Java3D-API erreicht wurde. Die Komponente `Szene3DModell` wurde zunächst in die Klasse `Universe` und die Klasse `BranchGroup` aufgesplittet. Die Klasse `Universe` hat die Aufgabe, einen Rahmen für die Darstellung der Informationen zu generieren und nimmt die berechnete Struktur auf, die von der Klasse `Scene` erzeugt wird. Die Klasse `BranchGroup` hingegen enthält die von der Klasse `Scene` berechnente Struktur. Bestandteil der `BranchGroup` ist `SimpleCube`, der für die graphische Darstellung von Klassen und Paketen sorgt.

Eine weitere Abweichung ist bei der Komponente `Navigation` zu finden. Hier ist in der implementierten Architektur die Kontrollrichtung zwischen `Szene3DModell` und `Navigation` umgekehrt, so dass also nun im `Universe` (ehemals `Szene3DModell`) eine Referenz auf den `KeyNavigatorBehavior` besteht. Auch dies ist durch das tiefere Verständnis der Java3D-API begründet. `KeyNavigatorBehavior` und `KeyNavigator` verfeinern die ursprüngliche Komponente `Navigation`.

Des Weiteren wird in der realisierten Architektur die Klasse `FrameOpener` von der Klasse `Controller` aufgerufen, um eine tatsächliche Darstellung des generierten Modells auf dem Bildschirm zu erhalten. Dies korrespondiert weitgehend mit der Systemmetapher. In der implementierten Architektur ist noch eine Klasse `Canvas3D` zu erkennen, die Java3D benötigt, um die entsprechenden Informationen dreidimensional zeichnen zu können.

Im Gegensatz zum ersten Entwurf wurde statt des *Eclipse*-Outline-Views ein eigener *Eclipse*-View erstellt. Dies liegt darin begründet, dass für den Outline-View ein interner *Eclipse*-Editor genutzt werden muss. Da von uns aber ein externes Swing-Fenster für die 3D-Darstellung genutzt wird, war die Eigenentwicklung nötig.

Ein letzter Unterschied ist bei der Speicherung des Startpunktes zu finden. Der Startpunkt wird in der realisierten Architektur in der Klasse `KeyNavigatorBehavior` gespeichert und nicht wie in der geplanten Architektur vorgesehen in der Komponente `Editor` (`Controller`). Auch diese Änderung ergab sich durch die im Laufe der Entwicklung gesammelten Kenntnisse über die Java3D-API.

14.5.4 Fazit

Zusammenfassend kann man sagen, dass die vorgegebene MVC-Architektur weitgehend beibehalten wurde. Im Wesentlichen wurde während der Entwicklung die Architektur weiter verfeinert. Unterschiede traten dabei aus mehreren Gründen auf: Zum einem steigerte sich die Kenntnis über die Java3D-API wie auch über die *Eclipse*-Plattform. Zum anderen wurden auch Strukturverbesserungen wie die Einrichtung der zentralen Steuerungsklasse `Controller` vorgenommen, um eine bessere Aufgabentrennung zu erreichen.

14.6 Kunden Akzeptanztest

René Schönlein

In der Sitzung vom 8.12.2003 fand der Akzeptanztest der Kunden an dem von den Entwicklern ausgelieferten Release-Kandidaten statt. Der Akzeptanztest hat den Zweck, die korrekte Umsetzung der in den User Stories spezifizierten Anforderungen zu überprüfen. Um diese Tests sinnvoll durchführen zu können, wurde von den Kunden eine Testumgebung vorbereitet und vorgestellt. Bei diesem Testprojekt (siehe Abb. 14.5, 14.6) handelt es sich um eine Klassenstruktur mit insgesamt 29 Klassen, die auf fünf Pakete verteilt sind, wobei die 29 Klassen in verschiedenen Vererbungsbeziehungen zueinander stehen. Die Tests, die von den Kunden durchgeführt wurden, lassen sich in zwei Kategorien einteilen. Zum einen in rein visuelle Tests, also Tests, die alleine durch das Betrachten der vom Plugin dargestellten Informationen erfolgen. Zum anderen in eine Kombination aus automatischen und visuellen Tests. Die automatischen Tests vergleichen hierbei erwartete Ergebnisse mit Ergebnissen, die vom Plugin bei der Ausführung generiert werden. Alle ausgeführten Tests spiegeln die Anforderungen in den von den Kunden verfassten User Stories wieder.

Hier nun eine Übersicht aller Tests die an dem Plugin ausgeführt wurden. Zu beachten ist, dass zuerst alle rein visuellen Tests stattfanden und dann die visuell-automatischen Tests.

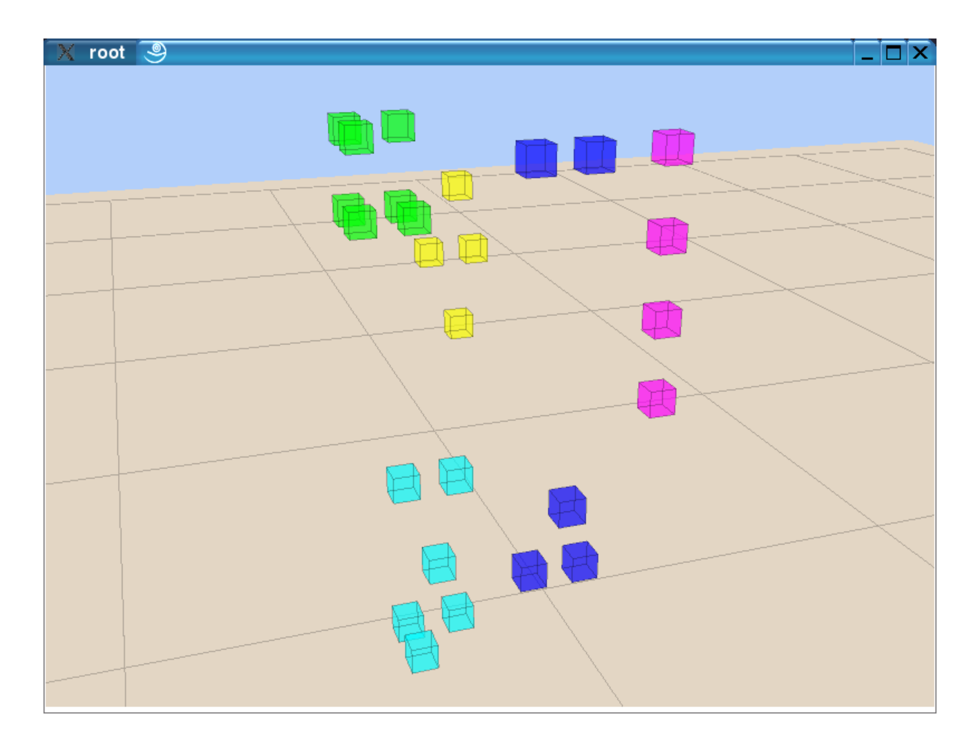


Abbildung 14.5.: Screenshot aus der Anwendung

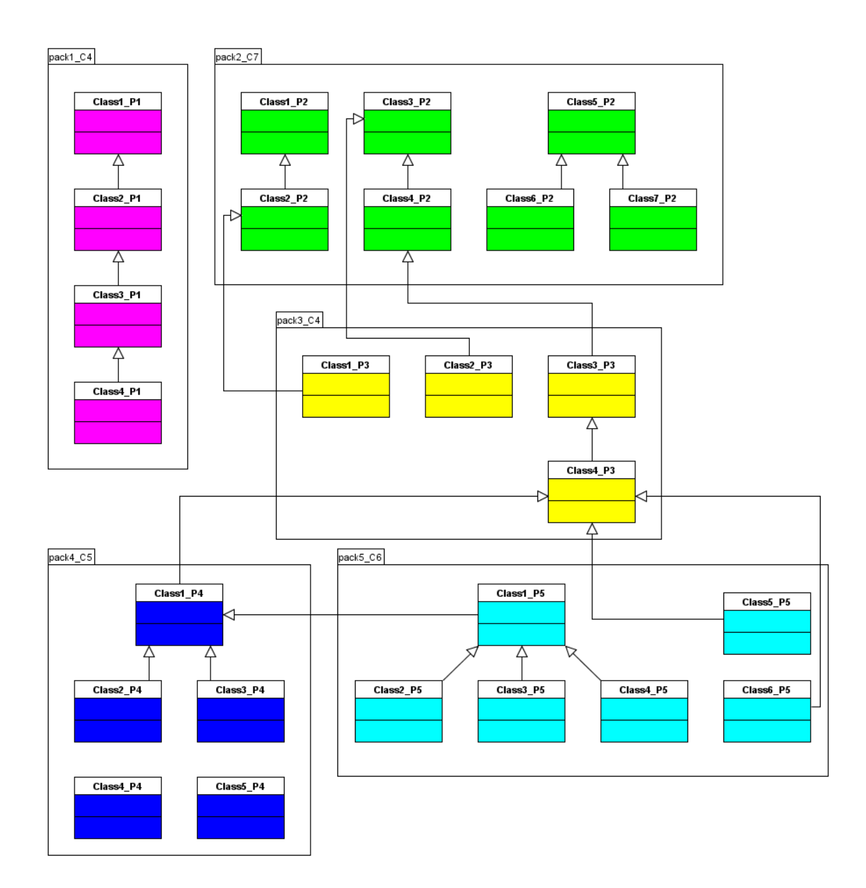


Abbildung 14.6.: Klassendiagramm des dargestellten Pakets

14.6.1 Visuelle Tests

Im Package-Explorer soll die Möglichkeit gegeben sein, ein ausgewähltes Paket in einem neuen externen Fenster visualisieren zu lassen.

Testergebnis: bestanden
Anmerkung: Es existiert nach Auswahl eines Pakets in dessen Kontextmenü ein neuer Menüpunkt mit Namen „Generate Scene", der das geforderte Verhalten anstößt.

Es ist möglich, in der 3D Szene zu navigieren.

Testergebnis: bestanden
Anmerkung: Es wird anhand der Pfeiltasten und der Tasten „Bild-Auf", „Bild-Ab" und „ALT" eine Navigation ermöglicht.

Klassen sollen als Würfel dargestellt werden.

Testergebnis: bestanden
Anmerkung: Durch das Navigieren in der Szene konnte verifiziert werden, dass die Repräsentation der Klassen als eindeutig zu erkennende Würfel erfolgt.

Klassen sollen geeignet beschriftet werden.

Testergebnis: bestanden
Anmerkung: Die Klassennamen erscheinen über den Würfeln. Hierbei ist allerdings zu beachten, dass die Klassennamen erst ab einer bestimmten Entfernung zum Bedrachter eingeblendet werden (vergl. Abb. 14.5).

Die Zuordnung der einzelnen Klassen zu einem Paket ist sofort durch ihre gehäufte Anordnung im Raum ersichtlich.

Testergebnis: bestanden
Anmerkung: Es wurde visuell von allen Seiten überprüft, ob zu erkennen ist, dass die zusammengehörenden Klassen auch räumlich nah beieinander stehen.

Es gibt eine Möglichkeit, die Sicht auf einen vorher definierten Startpunkt zurück zu setzen.

Testergebnis: bestanden
Anmerkung: Das geforderte Verhalten wird mittels Druck auf die Taste „H" erzielt.

Es existieren Informationsfenster, die Statusinformationen zur 3D-Ansicht enthalten.
Testergebnis: bestanden
Anmerkung: Unter Window → Show View → Other ... existiert nun ein neuer View mit Namen: ***EFFECTS***-Information-View“. In diesem View werden alle interessanten Informationen zur dargestellten 3D-Szene aufbereitet.

14.6.2 Auto-visuelle Tests

Die Würfel-Objekte (Klassen) werden bei der Szenengenerierung fest platziert.
Testergebnis: bestanden
Anmerkung: Durch das mehrmalige Aufrufen der 3D-Szenegenerierung konnte visuell überprüft werden, dass die Würfelobjekte immer an der gleichen Position platziert werden, solange sich an dem darzustellenden Projekt nichts ändert. Die automatische Überprüfung wurde mittels Speicherung der Würfel-Positionen in einer XML-Datei erreicht. Die XML-Datei wird dann bei einem erneuten Aufruf der Szenegenerierung mit den neuen internen Daten verglichen.

Kindklassen stehen immer auf einer graphisch tiefer liegenden Ebene (auch über Paketgrenzen hinaus) zu ihren Vaterklassen.
Testergebnis: bestanden
Anmerkung: Um die Korrektheit der dargestellten 3D-Szene zu überprüfen, wurden die Ebeneninformationen, die intern im Datenmodell gehalten werden, mit einer vorher vom Kunden spezifizierten XML-Datei abgeglichen. Die XML-Datei enthält die erwarteten Ebeneninformationen für jede darzustellende Klasse. Des Weiteren wurde die dargestellte Szene auch visuell überprüft.

Die Zuordnung der einzelnen Klassen zu einem Paket ist sofort durch ihre einheitliche Farbgebung ersichtlich.
Testergebnis: bestanden
Anmerkung: Es wurde visuell überprüft, ob alle Klassen eines Pakets die gleiche Farbe haben. Dieses wurde auch automatisch überprüft, indem verglichen wurde, ob die Farbe im `ClassContainer` einer Klasse mit der Farbe ihres Pakets übereinstimmt.

Es wird für jedes darzustellende Paket automatisch eine sinnvolle Farbe, die nicht zu ähnlich zu den bereits vergebenen Farben der anderen Pakete ist, ausgewählt.

Testergebnis: bestanden
Anmerkung: Es konnte visuell bestätigt werden, dass alle dargestellten Pakete eine unterschiedliche Farbe haben. Die ausgewählten Farben sind gut über das Farbspektrum verteilt. Es wurde automatisch überprüft, ob irgendeine Paketfarbe mehr als einmal verwendet wird.

Jede sinnvolle interne Information, z.B. Kameraposition oder Vaterklasse, muss über ein geeignetes Interface abfragbar sein (Methoden).
Testergebnis: bestanden
Anmerkung: Ohne geeignete Methoden wären die automatischen Tests nicht zu realisieren gewesen.

KAPITEL 15

Beschreibung des zweiten Release

15.1 Einleitung

Kai Gutberlet

Im zweiten Release soll die Architektur des ersten Releases zu einem Framework umgebaut werden. Des Weiteren soll das im ersten Release erstellte statische Diagramm an die neu entwickelte Frameworkstruktur angepasst werden. Dabei ist die Funktionalität des statischen Diagramms, die in den User Stories zum ersten Release in Kapitel 14.2 beschrieben ist, beizubehalten.

Im Folgenden wird der Aufbau des Frameworks und die Anpassung des statischen Diagrams an die neue Frameworkstruktur näher beschrieben.

15.2 User Stories

Kai Gutberlet, Michél Kersjes

Die im folgenden behandelten User Stories stellen die Anforderungsdefinition für das zweite Release des *Eclipse*-Plugins zur dreidimensionalen Visualisierung von Softwarestrukturen dar. Die Reihenfolge, in der sie aufgeführt sind, richtet sich nach der von den Kunden festgelegten Priorität.

Die wichtigste Aufgabe für das zweite Release bestand darin, das Plugin aus dem ersten Release strukturell zu einem Framework umzubauen. Die für diese Aufgabe wichtigen User Stories sind die HotSpots, über die das Framework benutzt werden kann. Die HotSpots sollen über eine XML-basierte Schnittstelle beschrieben werden, indem in einer Konfigurationsdatei angegeben wird, welche Klassen welchem HotSpot zugrunde liegen. Diese Klassen, deren Struktur durch Interfaces vorgegeben ist, implementieren z.B. für den Diagramm-HotSpot die graphischen Elemente.

Als erste Anwendung des Frameworks sollte die Funktionalität aus dem ersten Release realisiert werden. Diese Funktionalität ist die dreidimensionale Visualisierung von Javaklassen. Dazu wurde eine User Story definiert. Des Weiteren sollten neue Features in das zweite Release eingebracht werden, welche sich aus der Anwendung des Plugins aus dem ersten Release ergeben. Diese Features wurden ebenfalls als die folgenden User Stories verfasst: Die User Stories *Exception Handling* und *Berechnungsstatus* betreffen das Framework, wohingegen die User Stories *Paketauswahl* und *Schatten zur Orientierung* die dreidimensionale Visualisierung

von Javaklassen betreffen. Auf das gesamte Release bezieht sich die User Story *Deployment des Plugins*. Im Folgenden werden die einzelnen User Stories aufgelistet und kurz beschrieben.

Domain-HotSpot Beschreibung des Domains und seiner Entitäten über eine textuelle Schnittstelle. Dieser HotSpot beschreibt, welche Entitäten, wie z.B. Javaklassen, im Diagramm repräsentiert werden sollen.

Diagramm-HotSpot Beschreibung der visuellen Darstellung der Domainelemente über eine textuelle Schnittstelle. Hier wird die graphische Darstellung der Elemente festgelegt, wie z.B. die Darstellung als Würfel.

Regel-HotSpot Beschreibung der Diagrammregeln über eine textuelle Schnittstelle. Dabei bieten sich zwei mögliche Arten der Beschreibung an:

- **einfach:** Einfache Regeln legen die Zuordnung zwischen einem Element aus dem darzustellenden Domain und der graphischen Entität, durch die das Element dargestellt werden soll, fest. Ein Beispiel ist die Regel, dass Klassen als Würfel dargestellt werden sollen.

- **komplex:** Eine komplexe Regel beschreibt Zusammenhänge zwischen mehreren Elementen des Domains. Ein Beispiel ist, dass Vererbung durch die vertikale Anordnung der Würfel dargestellt werden soll.

User Stories des ersten Release Alle User Stories aus dem ersten Release müssen erfüllt bleiben. Die entsprechenden User Stories finden sich in der Beschreibung zum ersten Release.

Exception Handling Fehler sollen in der graphischen Benutzungsschnittstelle angezeigt werden.

Berechnungsstatus Während jeder Berechnung soll ein Informationsfenster angezeigt werden. Als Grafik für den Hintergrund des Fensters bietet sich das PG-Logo an. Innerhalb dieses Fensters ist eine Fortschrittsanzeige, die über den Status der Berechnungen Auskunft gibt, wünschenswert.

Deployment des Plugins Es soll eine Möglichkeit geboten werden, dass Plugin sowohl unter Linux als auch unter Windows auf einfache Weise zu installieren. Zu der Installation ist eine Dokumentation erwünscht.

Paketauswahl Es sollen mehrere Pakete gleichzeitig zur Visualisierung ausgewählt werden können, die anschliessend visualisiert werden.

Schatten zur Orientierung Die visuellen Diagrammelemente sollen zur besseren Orientierung ihren Schatten auf den Boden projizieren.

Alle hier aufgeführten User Stories wurden von den Entwicklern angenommen und in zugehörige Tasks aufgeteilt.

15.3 Systemmetapher

Semih Sevinç

Das Model-View-Controller-Konzept, das auch in der implementierten Architektur des ersten Releases Anwendung findet, hat sich als geeignet erwiesen. Daher wurde diese Systemmetapher auch für das zweite Release übernommen.

15.4 Reflexion über die Tasks

Armin Bruckhoff,Michael Pflug

Im Nachfolgenden wird über die Tasks des zweiten Release reflektiert. Hierzu werden die einzelnen Tasks kurz vorgestellt und insbesondere auf die signifikanten Probleme und die Differenzen in den Zeitabschätzungen eingegangen.

Die gegebenen User Stories wurden zunächst wie folgt als zwölf explizite Anforderungen aufgeschlüsselt und in einzelne Tasks unterteilt. Für jeden der Tasks wurde anschließend die Zeit abgeschätzt, die zur Umsetzung als nötig empfunden wurde.

15.4.1 Konzeptionelle Tasks bei den Hotspots

Unterschied zwischen Kernfunktionalität und erweiterten Funktionen

Beschreibung: Dieser Task hatte zur Aufgabe, die konzeptionellen Aufgaben für die entsprechende User Story zu erarbeiten und dabei herauszufinden, was Kernfunktionalität des Frameworks bzw. was die speziellen Eigenschaften der Erweiterungen sein sollen.

geplante Zeit: Die geplante Zeit betrug vier Stunden.

reale Zeit: Fünf Stunden. Dieser Task wurde zusammen mit dem nächsten Task „Plugin, Fragment oder Reflections für Erweiterungen“ durchgeführt. Da beide Tasks inhaltlich sehr ähnlich waren, wurde das Bearbeiten der Aufgabe zusammen ausgeführt. Die abgeschätzte Gesamtzeit von zwölf Stunden konnte dadurch so stark unterboten werden, dass die Analyse zum einem zu viert durchgeführt wurde. Zum anderen ließ sich die Trennung recht einfach (zumindest theoretisch) durchführen.

Plugin, Fragment oder Reflections für Erweiterungen

Beschreibung: Das Ziel dieses Tasks war es, eine Entscheidung darüber zu treffen, wie die Diagramme genau realisiert werden sollten. Dabei sollte festgelegt werden, wie zum einen die Definition der HotSpots (Domain, Regel, Diagramm) durchgeführt werden soll und zum anderen in welcher Form die Diagrammerweiterungen vorliegen werden.

geplante Zeit: Hier wurde bei der Planung zunächst berücksichtigt, dass einige technische Aspekte zu untersuchen sind, bevor über das weitere Vorgehen entschieden werden konnte. Daher wurde eine Zeit von acht Stunden eingeplant

reale Zeit: Siehe Task „Unterschied zwischen Kernfunktionalität und erweiterten Funktionen“.

Extension Points definieren

Beschreibung: Um einem Kernplugin weitere Funktionalität mittels eines angehängten Plugins oder Plugin-Fragments zur Verfügung zu stellen, mussten für das Kernplugin die entsprechenden Schnittstellen, in *Eclipse Extension Point* genannt, definiert werden. Diese Aufgabe sollte von diesem Task übernommen werden.

geplante Zeit: Die geplante Zeit betrug vier Stunden.

reale Zeit: Auch die benötigte Zeit betrug vier Stunden. Nach der Trennung der Kernfunktionalität von den Erweiterungen fiel es recht leicht, die entsprechenden Schnittstellen zu definieren.

Format für XML festlegen

Beschreibung: Eine weitere Anforderung dieses Release war es, eine Konfigurationsdatei im XML-Format zur Verfügung zu stellen. Damit soll es möglich sein, bestimmte Klassen – beispielsweise für verschiedene Darstellungen der Diagrammelemente – auszutauschen. Eine Definition der Datei bzw. des entsprechenden Schemas musste also gefunden werden.

geplante Zeit: Es wurde eine Zeit von fünf Stunden eingeplant.

reale Zeit: Für die tatsächliche Durchführung wurden nur zwei Stunden benötigt. Die Struktur der Datei ergab sich im Laufe der Arbeiten an den obigen konzeptionellen Tasks fast selbstständig, so dass an dieser Stelle lediglich die Ergebnisse zusammengefasst werden mussten und der Zeitaufwand somit gering gehalten werden konnte.

15.4.2 Domain-Hotspots

Implementierung der konzeptionellen Ergebnisse

Beschreibung: Dieser Task sollte, wie auch die Tasks „Diagramm-Hotspot“ und „Regel-Hotspot“, erledigt werden, sobald die Ergebnisse der Tasks zu den konzeptionellen User Stories vorliegen. Diese Ergebnisse sollten dann jeweils in entsprechenden User Stories je nach Kontext umgesetzt werden.

geplante Zeit: Da dieser Task auf den Ergebnissen der User Story „Konzeptionelle Tasks bei den Hotspots“ aufbaut und der Umfang der Arbeiten noch nicht abzusehen war, erfolgte hier keine Zeitabschätzung.

reale Zeit: Für die Durchführung der drei Hotspot-Tasks wurden insgesamt 16 Stunden benötigt. Das Refactoring war an einigen Stellen (z.B. Hinzufügen des Interface `Calculation` für die Berechnungsklassen) einfach zu lösen. An anderen Stellen hingegen war es komplizierter als erwartet. Insbesondere der `Controller` war nur schwer in den abstrakten Teil im Framework und den konkreten, im Plugin-Fragment implementierten Teil zu trennen. Darüber hinaus entstanden während der Umsetzung des dynamischen Ladens von Klassen aus zwei verschiedenen *Eclipse*-Plugins unerwartete Probleme. Es war nicht möglich, in einem Plugin Klassen dynamisch nachzuladen, die zu einem anderen Plugin gehören. Um dieses Problem zu lösen, wurde entschieden, das zunächst geplante Design zu ändern: Statt zwei Plugins zu entwickeln, wurde ein Plugin implementiert, das das Framework enthält. Die Diagramme werden als Plugin–Fragmente realisiert.

15.4.3 Diagramm-Hotspots

Implementierung der konzeptionellen Ergebnisse

Beschreibung: siehe Abschnitt 15.4.2 auf der vorherigen Seite
geplante Zeit: siehe Abschnitt 15.4.2 auf der vorherigen Seite
reale Zeit: siehe Abschnitt 15.4.2 auf der vorherigen Seite

15.4.4 Regel-Hotspots

Implementierung der konzeptionellen Ergebnisse

Beschreibung: siehe Abschnitt 15.4.2 auf der vorherigen Seite
geplante Zeit: siehe Abschnitt 15.4.2 auf der vorherigen Seite
reale Zeit: siehe Abschnitt 15.4.2 auf der vorherigen Seite

15.4.5 Alle User Stories aus Release 1 müssen erfüllt bleiben

Tests für alle Klassen hinzufügen

Beschreibung: Um alle Anforderungen an das erste Release zu erfüllen, müssen nachträglich noch alle Tests für jede existierende Klasse hinzugefügt werden.

geplante Zeit: Es wurde grob geschätzt, dass für 18 Klassen, zu denen es noch keine Testklassen gab, jeweils ca. zwei Stunden benötigt werden. So ergab sich dann eine geschätzte Zeit von 36 Stunden.

reale Zeit: Die reale Zeit betrug etwa die Hälfte der geschätzten Dauer. Dies lag daran, dass sich das Programmiererpaar nach einiger Zeit in das „Testschreiben“ eingearbeitet hatte und sich die Bearbeitungszeit pro Testklasse dementsprechend verringerte.

15.4.6 Fehler sollen in der GUI angezeigt werden (Exception Handling)

Fehlerdialog erstellen

Beschreibung: Hier war gefordert, einen Fehlerdialog zu erstellen, der alle auftretenden Exceptions in einem SWT–Dialog anzeigt.

geplante Zeit: Der Aufwand wurde bei diesem Task mit zwei Stunden angegeben.

reale Zeit: In ca. eineinhalb Stunden wurde der Task ohne besondere Vorkommnisse erledigt.

Benutzung des Fehlerdialogs sicherstellen (Exception-Handling überprüfen, Refactoring)

Beschreibung: Ein weiterer Task der User Story war es, die Benutzung des zuvor implementierten Fehlerdialogs zu gewährleisten. Es war nötig, sich an allen Codestellen zu vergewissern, dass auftretende Fehler an diesen Dialog weitergeleitet werden.

geplante Zeit: Die geschätzte Dauer betrug zwei Stunden, da im wesentlichen ein Überblick über den gesamten Code nötig war. Falls erforderlich, muss an bestimmten Stellen die Fehlerbehandlung angepasst werden.

reale Zeit: Der Task wurde in der Hälfte der geschätzten Zeit erledigt. Die Fehlerbehandlung war weitestgehend korrekt implementiert.

15.4.7 Deployment des Plugin

Erstellen eines Plugin sowohl für Linux als auch für Windows inklusive Dokumentation.

Beschreibung: Um das Plugin und das Plugin-Fragment korrekt sowohl unter Linux als auch unter Windows betreiben zu können, sollte eine Dokumentation erstellt werden, die das Erstellen zweier lauffähiger Versionen erläutert.

geplante Zeit: Diese Aufgabe wurde erst nach der Erstellung der Taskcards zu den User Stories mit aufgenommen. Daher erfolgte keine Einschätzung der Zeit.

reale Zeit: Der Task wurde in drei Stunden abgearbeitet. Ein Problem dabei war, dass für Windows zwei Versionen erstellt werden mussten, da es hier zwei Implementierungen der Java3D–Bibliothek gibt: eine für OpenGL und eine für DirectX.

15.4.8 Während jeder Berechnung soll ein Infofenster angezeigt werden (inkl. Logo). Eine Fortschrittsanzeige ist wünschenswert.

Statusfenster und Logo erstellen (in SWT)

Beschreibung: Um als Anwender erkennen zu können, ob bei der Generierung einer 3D-Szene der Vorgang aktiv ist oder nicht, soll ein Fenster erstellt werden, dass dieses signalisiert. Das Fenster soll dabei eine SWT-Komponente sein und das Logo der PG enthalten.

geplante Zeit: Die geplante Zeit betrug zwei Stunden.

reale Zeit: Die benötigte Zeit betrug zwei Stunden. Die Aufgabe bestand nur darin, das Design des Fensters zu erstellen.

Fortschrittsanzeige

Beschreibung: Neben dem zu erstellenden Fenster, das während des Ladevorganges angezeigt wird, soll eine Fortschrittsanzeige implementiert werden. Damit soll dem Benutzer signalisiert werden, dass das System Berechnungen durchführt und sich nicht in einem Deadlock befindet.

geplante Zeit: Die zeitliche Vorgabe lag für diesen Task bei zwei Stunden. Da die *Eclipse*-API bereits einen Fortschrittsbalken als Standardkonstrukt zur Verfügung stellt, wurde diese geringe Dauer als ausreichend erachtet.

reale Zeit: Auch hier mussten lediglich die entsprechenden SWT-Komponenten eingebunden werden.

15.4.9 Es sollen mehrere Pakete gleichzeitig zur Visualisierung ausgewählt und angezeigt werden können

Anpassen des Menüpunktes zur Mehrfachauswahl

Beschreibung: Ziel dieses Tasks war es, den Menüpunkt des Frameworks soweit anzupassen, dass es möglich ist, mehrere Pakete gleichzeitig auszuwählen und dann in der 3D-Sicht zusammen anzuzeigen.

geplante Zeit: Geplant wurde eine Stunde.

reale Zeit: Die geplante Zeit war ausreichend.

`Controller` anpassen

Beschreibung: Im Zusammenhang mit dieser User Story war es ebenfalls nötig, den Controller anzupassen, um die Funktion der Mehrfachauswahl zu ermöglichen. Da der `Controller` bisher nur genau ein `IPackageFragment` akzeptierte, mussten hier Veränderungen vorgenommen werden, so dass nun eine `Collection` von `IPackageFragments` verarbeitet werden kann.

geplante Zeit: Die geschätzte Dauer betrug zwei Stunden.

reale Zeit: Tatsächlich wurde für diesen Task ungefähr eine Stunde benötigt.

`DataModelHelper` für Mehrfachauswahl anpassen

Beschreibung: Um diese User Story zu erfüllen, war es abschließend noch nötig, die verschiedenen Hilfsfunktionalitäten des `DataModelHelpers` anzupassen. Dabei sollten alle bisherigen Hilfsfunktionen an die mehrfache Paketauswahl angepasst werden.

geplante Zeit: Für diesen Task wurde eine Zeit von etwa zwei Stunden eingeplant.

reale Zeit: Dieser Task wurde in ca. dreieinhalb Stunden erfüllt. Ein dabei aufgetretenes Problem war das Korrigieren der Auswahl. Es war erforderlich, dass keine Unterpakete eines Pakets, die ebenfalls selektiert sind, in der Auswahl sein dürfen. Da dies jedoch der Fall sein kann, musste hier mehr Zeit investiert werden, als zunächst eingeplant war.

15.4.10 Die visuellen Diagrammelemente sollen zur besseren Orientierung ihren Schatten auf den Boden werfen

Lichtquelle in das Universum einbauen

Beschreibung: Dieser Task sollte eine Lichtquelle zu der 3D-Szene hinzufügen, so dass alle auftretenden Objekte einen Schatten auf den Boden projizieren.

geplante Zeit: Die geplante Zeit betrug hier acht Stunden.

reale Zeit: Aus zeitlichen Gründen wurde diese Aufgabe nicht bearbeitet. Die Erledigung der User Story „Schattenwurf“ wird in das nächste Release verschoben.

15.4.11 Fazit

Zusammenfassend kann gesagt werden, dass fast alle zu erledigenden Aufgaben auch erledigt wurden. Lediglich die Implementierung der Objektschatten wurde in diesem Release nicht realisiert. Für die konzeptionellen Aufgaben wurde insgesamt zu viel Zeit veranschlagt, für die Hotspots wurde zunächst keine Zeitabschätzung gemacht, da die Aufgaben aus den Ergebnissen der konzeptionellen Arbeiten zu nehmen waren. Damit wurde die eingesparte Zeit aus den konzeptionellen Tasks wieder verbraucht, so dass insgesamt die Zeitabschätzung mit der tatsächlich benötigten Zeit im Wesentlichen übereinstimmte.

15.5 Vorstellung der implementierten Architektur

André Kupetz, Stephan Eisermann, Daniel Unger, Jan Wessling

Im Nachfolgenden soll die in diesem Release implementierte Architektur vorgestellt werden. Dazu wird zunächst die geplante Architektur beschrieben, im Anschluss daran die realisierte Architektur festgehalten und abschließend beide miteinander verglichen.

15.5.1 Beschreibung der geplanten Architektur

Die geplante Architektur für das zweite Release ist in Abbildung 15.1 auf Seite 151 dargestellt. Es werden im Folgenden die entscheidenden Unterschiede zur Realisierung des ersten Releases erläutert.

Das MVC-Konzept wurde beibehalten. Allerdings ergaben sich durch die neuen Anforderungen des zweiten Releases eine Aufteilung des Plugins in ein Kernplugin und optional mehrere diagrammtypspezifische Plugins. Der Kernbereich umfasst nur noch die Klassen, die für jeden Diagrammtyp gleich sind. Dies sind die Klassen, die nichts mit der konkreten Visualisierung, dem Datenmodell sowie den Berechnungsvorschriften des Diagrammtyps zu tun haben. So sind die Navigation und die Modellgrundlagen (`IType`, `IResource`, `ICompilationUnit`) immer gleich. Weiterhin gibt es immer eine `Scene` sowie das dazugehörige `Universe`.

Der Kern stellt weiterhin je ein Interface für Diagrammelemente (`GraphicalObject`), Berechnungen (`Calculation`) und Modellinformationen (`Container`) zur Verfügung. Diese drei Interfaces dienen als Schnittstelle für die konkreten Diagrammplugins. Das Interface `Container` stellt den geforderten Domain-Hotspot bereit. Hier werden die Elemente festgelegt, welche in dem jeweiligen Diagrammtyp dargestellt werden sollen.

Für den im ersten Release implementierten Diagrammtyp beispielsweise sind diese Elemente Klassen und Pakete. Das Interface `GraphicalObject` stellt den Diagramm-Hotspot zur Verfügung, in dem die graphische Darstellung festgelegt wird. Für das obige Beispiel des dreidimensionalen Klassendiagramms wird im Diagramm-Hotspot festgelegt, dass Klassen als Würfel dargestellt werden. Der Regel-Hotspot schließlich wird vom Interface `Calculation` bereitgestellt. Hier werden die Regeln zur Anordnung der graphischen Elemente in der Darstellung festgelegt. Im Beispiel sind solche Regeln etwa die Ebenenanordnung der Klassen

nach ihrer Position in der Vererbungshierarchie oder die räumlich nahe Anordnung von Klassen eines Pakets.

15.5.2 Beschreibung der realisierten Architektur

In diesem Release wurden die Anforderungen so erweitert, dass verschiedene Diagrammtypen eingebunden werden können. Diese Anforderungen führten auch zu grundlegenden Veränderungen in der Architektur. Diese besteht jetzt aus einem Kernplugin und aus den für die Diagramme spezifischen Plugin-Fragementen. Die implementierte Architektur wird in Abbildung 15.2 auf Seite 152 gezeigt.

Zum Kernplugin gehören nur noch die Klassen, die allgemeine Funktionalität bereitstellen, welche für alle Diagrammtypen gleich sind. Dazu gehören die Klassen, die als Grundlage für das Datenmodell dienen und die aus dem ersten Release übernommen wurden. Es sind die von *Eclipse* bereitgestellten Klassen `IType`, `ICompilationUnit` sowie das Interface `IResource`. Ebenfalls übernommen wurden die Klassen, die für die Darstellung einer Szene verantwortlich sind. Dazu zählen die Klassen `Universe` und `ViewController` sowie die von Java3D bereitgestellten Klassen `Canvas3D` und `BranchGroup`. Die Navigation durch die 3D-Darstellung ermöglichen weiterhin die Klassen `KeyNavigator` und `KeyNavigatorBehaviour`. Die für die Berechnung zuständige Klasse `Scene` ist zu einer abstrakten Klasse geworden, da die Berechnung jetzt für die verschiedenen Diagrammtypen unterschiedlich verläuft.

Um eine Schnittstelle zu den entsprechenden Plugins der Diagrammschnittstelle zu haben, stellt das Kernplugin verschiedene Interfaces bereit. Zur Speicherung der Informationen der entsprechenden Diagrammtypen wird ein Interface `Container` zur Verfügung gestellt. Um diese Information graphisch darzustellen, existiert das Interface `GraphicalObject`. Schließlich gibt es für die Festlegung der zu verwendenden Regeln das Interface `Calculation`.

Um die entsprechenden diagrammtypspezifischen Plugins in das Kernplugin einzuhängen, gibt es Extension Points. In diesem Release sind die beiden Klassen `Controller` und `InformationView` die Extension Points. Deren Funktionalität hat sich gegenüber dem ersten Release allerdings nicht geändert. Der `Controller` dient weiterhin zum Aufruf der einzelnen Berechnungen und zum Aufruf der Visualisierung. Der `InformationView` ist für die Darstellung wichtiger Informationen in einem eigenen Fenster zuständig. Als Verfeinerung gegenüber dem ersten Release sind die Klassen `EffectsXMLParser` und `DiagramConfig` neu hinzugekommen.

15.5.3 Vergleich zwischen geplanter und realisierter Architektur

Allgemein lässt sich sagen, dass die Unterschiede zwischen der geplanten Architektur und der tatsächlich implementierten Architektur nicht so gravierend sind, wie es im ersten Release der Fall war. Die für die Visualisierung zuständigen Klassen wurden fast vollständig so implementiert, wie sie geplant wurden. Der Zugriff auf die darstellenden Klassen wird komplett über die Klasse `ViewController` realisiert. Dafür ist die Klasse `FrameOpener` komplett weggefallen und aus der Klasse `Scene` wurde eine abstrakte Klasse. Dass die Visualisierung

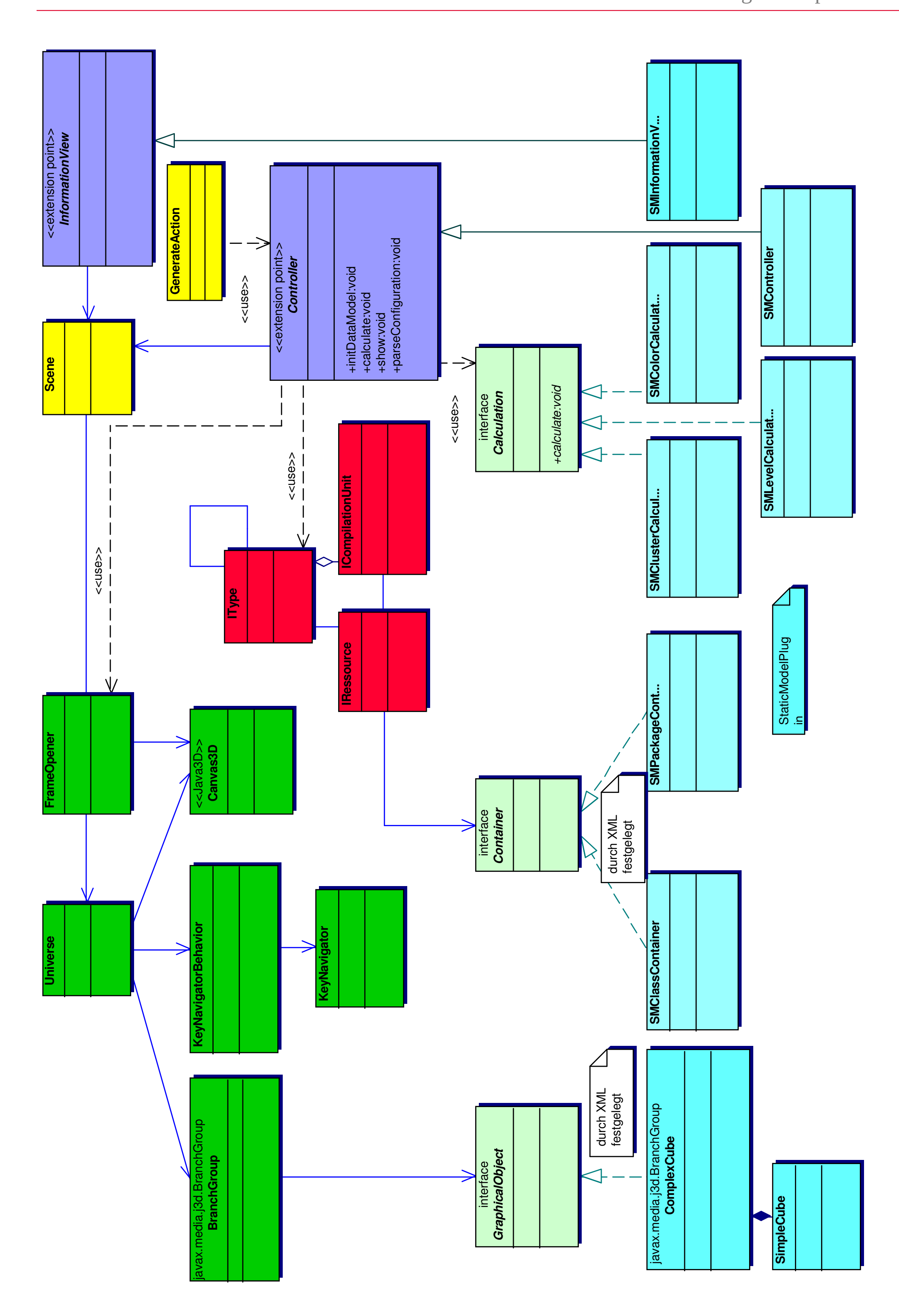


Abbildung 15.1.: Geplante Systemarchitektur Release 2

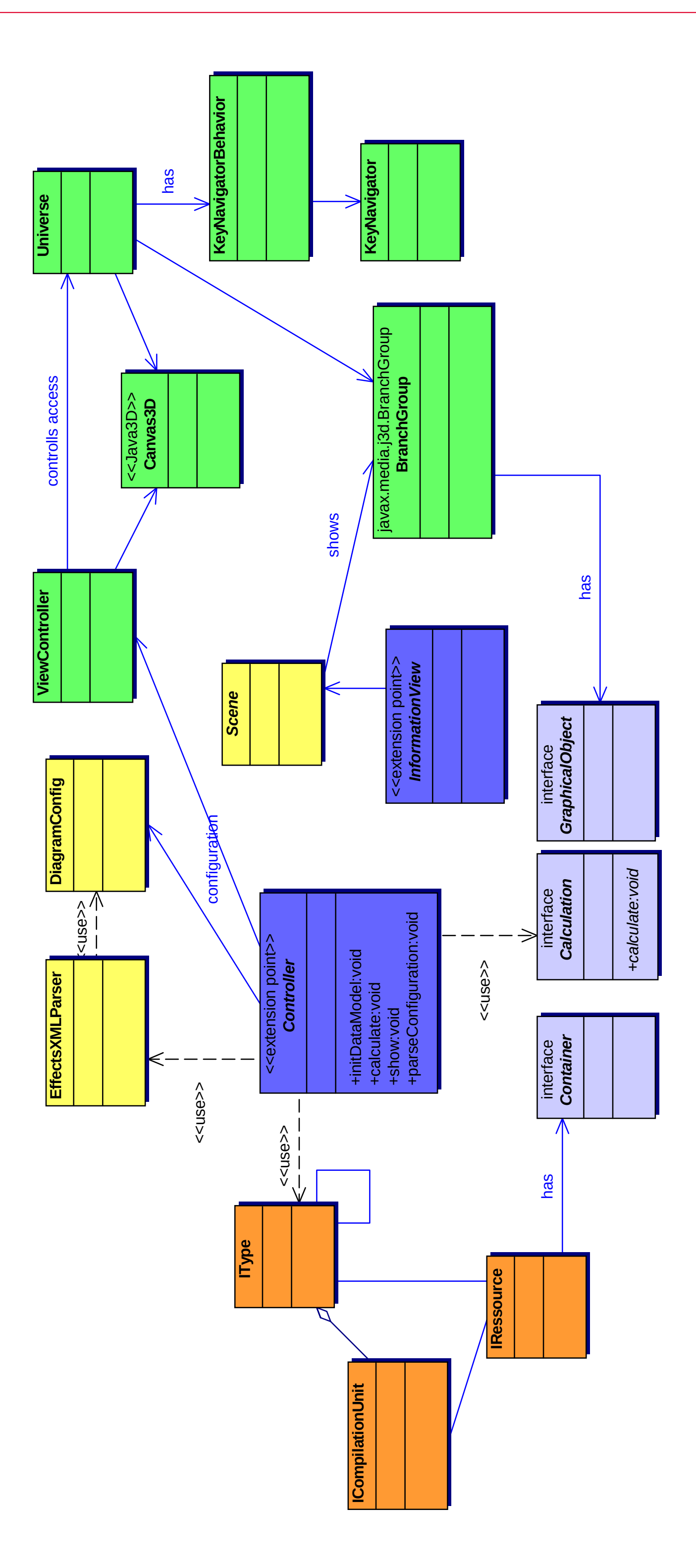


Abbildung 15.2.: Implementierte Systemarchitektur Release 2

bis auf wenige Veränderungen so implementiert wurde wie sie geplant war, liegt auch darin begründet, dass sie auf der Visualisierung des ersten Releases aufbaut. Die Darstellung der Informationen wurde wie geplant implementiert. Dabei erfragt die Klasse `InformationView` die anzuzeigenden Daten von der abstrakten Klasse `Scene`. Auch die Klassen, die als Grundlage für das Datenmodell dienen, wurden so realisiert, wie sie geplant waren. Es handelt sich um die von *Eclipse* bereitgestellten Klassen `ICompilationUnit` und `IType` sowie das Interface `IResource`. Auch diese wurden bereits im ersten Release verwendet.

Um ein diagrammtypspezifisches Plugin in das Kernplugin einzuhängen, müssen die durch das Kernplugin bereitgestellten `DiagramGeneration` und `InformationViewAddition` genutzt werden. In der Manifestdatei des Fragments müssen den genutzten Extension Points noch Klassen zugeordnet werden. In diesem Release wurden die bereits erwähnten Klassen `InformationView` und `Controller` den Extension Points zugeordnet. Dieses entspricht ebenfalls der geplanten Architektur. Die Funktionalität dieser Klassen hat sich gegenüber dem ersten Release allerdings nicht grundlegend verändert.

In diesem Release wurde eine Aufteilung in ein Plugin, welches die Kernfunktionalitäten enthält, die für alle Diagrammtypen benötigt werden, und in die entsprechenden diagrammspezifischen Plugins vorgenommen. Dabei muss das Kernplugin Schnittstellen bereitstellen, die als Hotspots zu den diagrammtypspezifischen Plugins dienen. Das Interface `Container` dient dabei als Domain-Hotspot. Hier sollen die Informationen gehalten werden, die für die Darstellung eines speziellen Diagramms benötigt werden. Das Interface `Calculation` stellt den Regel-Hotspot bereit. Hier werden die Regeln, die zur Anordnung der Elemente eines Diagrammtyps in der 3D-Darstellung benötigt werden, definiert. Das dritte Interface ist das Interface `GraphicalObject`, welches als Diagramm-Hotspot dient. Er stellt die graphische Darstellung der einzelnen Elemente in der 3D-Szene bereit. Auch diese Interfaces wurden so implementiert, wie sie in der geplanten Architektur vorgestellt wurden. Eine Änderung gegenüber der geplanten Architektur stellen die beiden Klassen `EffectsXMLParser` und `DiagramConfig` dar. Diese waren in der geplanten Architektur nicht vorgesehen und sind erst im Laufe der Implementierung neu entstanden.

15.6 Kunden Akzeptanztest

Kai Gutberlet, Michél Kersjes

Am 19. Januar 2004 fand der Akzeptanztest für das zweite Release statt. Im Folgenden werden die User Stories mit den geplanten Akzeptanztests für das zweite Release nach ihrer Durchführung geordnet aufgeführt. Alle User Stories aus dem ersten Release müssen erfüllt bleiben.

15.6.1 Durchgeführte Kundentests

Im *Package-Explorer* soll die Möglichkeit gegeben sein, ein ausgewähltes Paket in einem neuen, externen Fenster visualisieren zu lassen.

Testergebnis: bestanden

Anmerkung: manuelle Überprüfung: Es soll ein Paket ausgewählt und über einen Menüpunkt das Plugin gestartet werden.

Es soll möglich sein, in der dreidimensionalen Szene zu navigieren.
Testergebnis: bestanden
Anmerkung: manuelle Überprüfung: mittels Tastatur in der dreidimensionalen Szene navigieren

Klassen sollen als Würfel dargestellt werden.
Testergebnis: bestanden
Anmerkung: visuelle Überprüfung

Klassen sollen geeignet beschriftet werden.
Testergebnis: bestanden
Anmerkung: visuelle Überprüfung

Die Zuordnung der einzelnen Klassen zu einem Paket soll sofort durch ihre gehäufte Anordnung im Raum ersichtlich sein.
Testergebnis: bestanden
Anmerkung: visuelle Überprüfung

Es soll eine Möglichkeit geben, die Sicht auf einen vorher definierten Startpunkt zurückzusetzen.
Testergebnis: bestanden
Anmerkung: visuelle Überprüfung

Es sollen Informationsfenster existieren, die Statusinformationen zur 3D-Ansicht enthalten.
Testergebnis: bestanden
Anmerkung: manuelle Überprüfung: unter Window → Show View → Other ... den neuen View des Plugins aufrufen.

Die Würfelobjekte (Klassen) sollen bei der Szenegenerierung fest platziert werden.
Testergebnis: bestanden

Anmerkung: automatische und visuelle Überprüfung: Mehrmaliges Aufrufen der Generierung dreidimensionaler Szenen und Vergleichen der Darstellung. Die automatische Überprüfung wird mittels Speicherung der Würfelpositionen in einer XML-Datei erreicht. Die XML-Datei wird dann bei einem erneuten Aufruf der Szenegenerierung mit den neuen internen Daten verglichen.

Kindklassen sollen immer auf einer graphisch tiefer liegenden Ebene (auch über Paketgrenzen hinaus) als ihre Vaterklassen stehen.

Testergebnis: bestanden

Anmerkung: automatische und visuelle Überprüfung: Um die Korrektheit der dargestellten 3D-Szene zu überprüfen, werden die Ebeneninformationen, die intern im Datenmodell gehalten werden, mit einer vorher vom Kunden spezifizierten XML-Datei abgeglichen. Die XML-Datei enthält die erwarteten Ebeneninformationen für jede darzustellende Klasse.

Die Zuordnung der einzelnen Klassen zu einem Paket soll sofort durch ihre einheitliche Farbgebung ersichtlich sein.

Testergebnis: bestanden

Anmerkung: automatische und visuelle Überprüfung: Es wird verglichen, ob die Farbe im `ClassContainer` einer Klasse mit der Farbe ihres Pakets übereinstimmt.

Es soll für jedes darzustellende Paket automatisch eine sinnvolle Farbe ausgewählt werden, die nicht zu ähnlich zu den bereits vergebenen Farben der anderen Pakete ist.

Testergebnis: bestanden

Anmerkung: automatische und visuelle Überprüfung: Es wird automatisch überprüft, ob irgendeine Paketfarbe mehr als einmal verwendet wird.

Sinnvolle interne Informationen, z.B. Kameraposition oder Vaterklasse, sollen über ein geeignetes Interface abfragbar sein (Methoden).

Testergebnis: bestanden

Anmerkung: Ohne geeignete Methoden wären die automatischen Tests nicht zu realisieren gewesen

Beschreibung der Domain und ihrer Entitäten über eine textuelle Schnittstelle.

Testergebnis: bestanden
Anmerkung: manuelle Überprüfung: In der XML-Datei kann über das Tag `Container` das *Eclipse*-Datenmodell erweitert werden. Das für dieses Release erstellte `StaticModel`-Plugin stellt bereits einen Test für den Domain HotSpot dar, da die Domain dieses Plugins ein statisches Klassendiagramm ist. Um für diese User Story einen weiteren Test durchführen zu können, müsste ein Diagramm für einen anderen Anwendungsbereich erstellt werden.

Beschreibung der visuellen Darstellung der Domainelemente über eine textuelle Schnittstelle.
Testergebnis: bestanden
Anmerkung: manuelle Überprüfung: Konfigurieren der XML-Datei, indem die einen Würfel implementierende Klasse `ComplexCube` durch die einen Tetraeder implementierende Klasse `ComplexTriangle` ausgetauscht wird. Anschließend wird überprüft, ob nun ein Tetraeder als Symbol für eine Klasse angezeigt wird.

Beschreibung der Diagrammregeln über eine textuelle Schnittstelle.
Testergebnis: bestanden
Anmerkung: manuelle Überprüfung: Die Regel zur Vergabe von Farben für Pakete ist in der Klasse `ColorCalculations` implementiert. Daher wird die XML-Datei geändert, indem der Verweis auf die das Interface `Calculations` implementierende Klasse `ColorCalculations` entfernt wird. Anschließend wird überprüft, ob die Elemente im Diagramm farblos erscheinen.

Fehler sollen in der graphischen Benutzungsschnittstelle angezeigt werden.
Testergebnis: nicht bestanden
Anmerkung: manuelle Überprüfung: In der XML-Datei wird ein Verweis auf eine Klasse eingetragen, die einen Fehler verursacht. Dieser muss während der Programmausführung angezeigt werden. Des Weiteren wird in der XML-Datei ein Verweis auf eine nicht existierende Klasse eingetragen. Auch dieser Fehler soll in der graphischen Benutzungsschnittstelle angezeigt werden. Der Test wurde nicht bestanden, da bei einem provozierten Fehler bezüglich des Regel-HotSpot keine Meldung in der graphischen Benutzungsschnittstelle angezeigt wurde.

Während jeder Berechnung soll ein Informationsfenster angezeigt werden, dass wünschenswerterweise eine Fortschrittsanzeige enthält.
Testergebnis: bestanden
Anmerkung:

1. visuelle Überprüfung
2. manuelle Überprüfung: Die für den Test geschriebenen Methoden werden nach Abschluß der Berechnung aufgerufen. Sobald diese Methoden aufgerufen werden, soll das Informationsfenster nicht mehr angezeigt werden.

Es sollen mehrere Pakete gleichzeitig zur Visualisierung ausgewählt und angezeigt werden können.
Testergebnis: bestanden
Anmerkung: automatische und visuelle Überprüfung: Es werden mehrere Pakete ausgewählt und das Plugin über den Menüpunkt gestartet. Anschließend wird in einem Testdialog eine von den Kunden erstellte XML-Datei ausgewählt, die der Paketauswahl entspricht. Nun werden die berechneten Daten automatisch mit den erwarteten Daten verglichen.

Die visuellen Diagrammelemente sollen zur besseren Orientierung ihren Schatten auf den Boden projizieren.
Testergebnis: nicht bestanden
Anmerkung: visuelle Überprüfung: Dieser Test konnte nicht durchgeführt werden, da die User Story aus Zeitgründen nicht implementiert wurde.

Es soll eine Möglichkeit geboten werden, das Plugin sowohl für Linux als auch für Windows auf einfache Weise zur Verfügung zu stellen. Dazu ist eine Dokumentation erwünscht.
Testergebnis: nicht bestanden
Anmerkung: manuelle Überprüfung: Die für das Plugin zur Verfügung gestellten Installationsdateien werden gemäß Dokumentation installiert. Anschließend wird *Eclipse* gestartet und überprüft, ob das Plugin zur Verfügung steht und korrekt funktioniert. Dieser Test schlug fehl, da bei den erstellten Plugins für Windows nicht berücksichtigt wurde, das Java3D für Windows in einer Version für *DirectX* und *OpenGL* vorliegt.

15.6.2 Fazit

Nach Durchführung aller Akzeptanztests wurde das zweite Release trotz kleiner Mängel von den Kunden sowie der Geschäftsleitung abgenommen. Die wichtigste Anforderung, das Plugin strukturell zu einem Framework umzubauen, wurde erfolgreich erfüllt. Kleinere Mängel beziehen sich auf das Deployment des Plugins unter Windows, auf die Fehleranzeige in der graphischen Benutzungsschnittstelle und auf die Projektion von Schatten. Bei dem Deployment wurde nicht berücksichtigt, dass für *OpenGL* und *DirectX* jeweils unterschiedliche Bibliotheken verwendet werden. Der Mangel der Fehleranzeige ist es, dass nicht alle Fehler korrekt in der graphischen Benutzungsschnittstelle angezeigt werden. Des Weiteren konnte aus Zeitgründen der Task zur Projektion von Schatten nicht bearbeitet werden. Da dieser Task jedoch mit der niedrigsten Priorität eingestuft war, fällt dieser Mangel gegenüber den beiden anderen nicht so sehr ins Gewicht. Eine Überarbeitung der fehlerhaften Tasks soll parallel zur Entwicklung des dritten Releases stattfinden.

KAPITEL 16

Beschreibung des dritten Release

16.1 Einleitung

Daniel Unger, Semih Sevinç

Ziel dieses Releases war es, einen neuen Diagrammtyp zu erstellen. Als Grundlage hierfür diente das 3D-Sequenzdiagramm von (Gil und Kent, 1998) . Es handelt sich dabei um eine Kombination aus Sequenz -und Kollaborationsdiagramm in dreidimensionaler Darstellung. Ein Sequenzdiagramm stellt den konkreten Ablauf eines Anwendungsszenarios unter Einbeziehung der beteiligten Objekte dar. Durch die Betonung auf den zeitlichen Ablauf, wird der Nachrichtenaustausch der Objekte leicht ersichtlich. Ein Kollaborationsdiagramm zeigt die gleichen Sachverhalte wie ein Sequenzdiagramm, jedoch aus einer anderen Perspektive. Bei diesem dynamischen Diagrammtyp stehen die Objekte und ihre Zusammenarbeit untereinander im Vordergrund und nicht der zeitliche Ablauf. Dahingegen wird beim Sequenzdiagramm nicht auf die Kollaboration von Objekten eingegangen. Eine Kombination dieser Interaktionsdiagramme führt die Aspekte der einzelnen Diagrammtypen zusammen.

Der Aufbau des neuen dreidimensionalen Diagrammtyps wird jetzt im Folgenden näher erläutert. Das dreidimensionale Interaktionsdiagramm besteht aus mehreren Ebenen, auf denen die Objekte analog zum Kollaborationsdiagramm nach der Häufigkeit ihrer Interaktion angeordnet werden. Objekte, welche oft miteinander interagieren, liegen entsprechend nahe zusammen. Wenn ein neues Objekt erzeugt wird, wird eine neue Ebene angelegt, auf dem das Objekt als Quader gezeichnet wird. Jedes Objekt hat eine Lebenslinie, die durch alle Ebenen hindurch bis auf den Boden gezeichnet wird. Die Aktivität eines Objektes wird durch einen Aktivitätsbalken in Form eines Zylinders um die Lebenslinie dargestellt. Interagieren zwei Objekte miteinander, so wird dieses mittels eines Pfeils zwischen den Ebenen vom aufrufenden Objekt zum aufgerufenen Objekt dargestellt. Im Gegensatz zum 3D-Sequenzdiagramm von Gil und Kent, haben wir bei unserem Diagrammtyp die Darstellung der benutzen Parameter mittels einer Verbindung zwischen den Objekten auf einer Ebene der Einfachheit halber weggelassen.

16.2 User Stories

Jan Wessling, Christian Mocek

Die im Folgenden aufgeführten User Stories stellen die Anforderungsdefinition für das dritte Release des *Eclipse*-Plugins zur dreidimensionalen Visualisierung von Softwarestrukturen dar.

Die Aufgabe des dritten Releases war es, einen neuen Diagrammtyp zu implementieren. Er sollte nach der Semantik von (Gil und Kent, 1998) gestaltet werden. Eine Beschreibung des zu entwickelnden Diagrammtypes befindet sich in Kapitel 16.1.

Diese User Stories wurden komplett von den Entwicklern angenommen:

Popup-Menü

Im Package-Explorer soll ein eigener Menüpunkt ***EFFECTS*** in das Popup-Menü eingefügt werden. In dieses Menü soll es für jeden Diagrammtypen einen Eintrag geben, der die Generierung eines entsprechenden Diagramms startet.

Schattenwurf

Es sollen Lichtquellen eingebunden werden, so dass die Objekte Schatten auf den Boden werfen.

Integration von Java 3D in *Eclipse*

Das Ausgabefenster soll als *Eclipse*-Editor implementiert werden. Es soll also kein AWT-Fenster mehr für ein Diagramm geöffnet werden, sondern das Diagramm soll sich in der *Eclipse*-Oberfläche integrieren.

Tests

Sämtliche vorhandene Tests müssen auf die neue Framework-Systematik angepasst werden.

Screenshot

Es soll möglich sein, Screenshots von der aktuellen Ansicht zu generieren und zu speichern.

EFFECTS-Perspektive

Es soll eine eigene *Eclipse*-Perspektive für das ***EFFECTS***-Framework generiert werden. Zu dieser Perspektive ist ein eigenes Pull-Down Menü für ***EFFECTS*** in die Menüleiste hinzuzufügen.

Steuerung

Die Steuerung soll auf Maus und Tastatursteuerung angepasst werden, ähnlich wie sie in sog. Ego-Shootern verwendet wird.

Hilfe

Die Hilfe zu den Extension Points und ein Tutorial zur Implementierung eigener Diagrammtypen sollen als Online-Hilfe in *Eclipse* eingebunden werden.

Level-of-Detail

Objekte sollen ab einer bestimmten Distanz zur Kamera (optional) automatisch ausgeblendet werden können.

Kamerafahrt

Auf der Zeitachse des Diagramms soll eine automatische Kamerafahrt möglich sein. Wegpunkte müssen nicht frei bestimmt werden können.

Sequenzdiagramme

Grundlage Für das Diagramm soll die Semantik von Gil und Kent, wie in der Einleitung beschrieben, dienen, jedoch ohne Objektzustände.

Erzeugung Eine automatische Generierung des Diagramms aus dem zugrundeliegenden Code ist gewünscht.

Syntaxunterstützung Es soll möglich sein, bei der Erzeugung eines neuen Diagramms das aktuelle Projekt als Basis zu verwenden.

Ein Objekt soll als flacher Quader angelegt werden, wobei von diesem Quader eine Lebenslinie auf der Zeitachse einzuzeichnen ist. Neue Objekte erscheinen erst auf der Zeitachse, wenn sie erzeugt werden.

Aktivitätsbalken Die Aktivitätsbalken sollen als Röhre um die Lebenslinie herum dargestellt werden.

Ebenen Pro Methodenaufruf ist eine Ebene erforderlich, auf der neben dem Methodenaufruf auch noch der Parameteraufruf mittels eines Pfeils einzuzeichnen ist. Die beiden Pfeilarten sollen unterschiedliche Farben besitzen.

16.3 Systemmetapher

Semih Sevinç

Zur Entwicklung des neuen Diagrammtyps, wurde die Framework-Architektur des zweiten Release (Kapitel 15.5.2) weiterhin benutzt. Da das Framework nur um ein Plugin für den neuen Diagrammtyp erweitert werden sollte, konnte die selbe Systemmetapher ebenfalls weiterhin verwendet werden. Als Grundlage für die Vorgehensweise bei der Implementierung wurde damit, wie bereits bei den ersten beiden Releases auch, das Model-View-Controller-Konzept als Systemmetapher benutzt. Dieses Konzept lässt sich in drei Bereiche einteilen. Das Modell stellt die Daten einer Anwendung dar, kann deren Zustand liefern und Veränderungen an ihnen vornehmen. Der View ist für die graphische Darstellung der Daten zuständig und dient somit als visuelle Darstellung des Modells. Der Zugriff des Views auf das entsprechende Modell wird durch den Controller definiert. Im Abschnitt 14.3 wird gezeigt, wie diese einzelnen Bereiche in unserem Projekt umgesetzt worden sind.

16.4 Reflexion über die Tasks

Daniel Unger,Michael Pflug

Im Nachfolgenden wird über die Tasks des dritten Release reflektiert. Hierzu werden die einzelnen Tasks kurz vorgestellt und insbesondere auf die signifikanten Probleme und die Differenzen in den Zeitabschätzungen eingegangen.

Die gegebenen User Stories wurden zunächst wie folgt als elf explizite Anforderungen aufgeschlüsselt und in einzelne Tasks unterteilt. Für jeden der Tasks wurde anschließend die Zeit abgeschätzt, die zur Umsetzung als nötig empfunden wurde.

16.4.1 Popup-Menü

Menüpunkt *EFFECTS* im Package-Explorer

Beschreibung: Das Ziel dieses Tasks war es, einen eigenen ***EFFECTS***-Menüpunkt im Kontextmenü des Package-Explorers zu schaffen. Die Funktionalität des entsprechenden ***EFFECTS***-Plugins soll nun immer unter diesem Menüpunkt zu finden sein.

geplante Zeit: Eine Stunde wurde veranschlagt, da ein Menüpunkt ja schon vorhanden war (aus den Releases davor), und nur eine neue Hierarchie eingeführt werden sollte.

reale Zeit: Der Task wurde wie zeitlich geplant erfüllt.

16.4.2 Schattenwurf

Lichtquellen sollen Schatten projizieren

Beschreibung: Dieser Task wurde schon aus den vorherigen Releases hierher verlagert. Aus Zeitgründen und aufgrund der niedrigen Priorität wurde dieser Task allerdings wieder nicht bearbeitet. Jedoch sind Lichtquellen und Schatten für den in diesem Release entwickelten Sequenzdiagrammtyp auch nicht sonderlich relevant. Sie würden keine wichtigen, zusätzlichen Informationen zur Anordnung der Objekte im Raum liefern, wie es beim statischen Modell der Fall wäre.

geplante Zeit: Zehn Stunden wurden für die Bearbeitung geschätzt.

reale Zeit: Null Stunden, da der Task aus zeitlichen Gründen nicht erledigt wurde.

16.4.3 Integration von Java 3D in *Eclipse*

Editor anlegen

Beschreibung: Aufgabe sollte sein, ein Fenster bzw. Editor in *Eclipse* anzulegen, so dass die Anzeige der generierten Szene direkt in diesem Editor in *Eclipse*, statt in einem externen Fenster möglich wäre.

geplante Zeit: Geschätzte Zeit vier Stunden.

reale Zeit: Im Wesentlichen war dieser Task kein Problem, da das Wissen einen Editor zu erstellen, bei der Erstellung des Viewprototypen gewonnen wurde. Einziges Problem war das Abspeichern und Schließen des Editors, da vergessen wurde, dem Editor den `IEditorInput` bekannt zu machen. Dieser stellt den Inhalt des Editors dar und wird für das Speichern und Schließen benötigt. Diese Aufgabe konnte somit sogar in nur drei Stunden erledigt werden.

AWT in SWT

Beschreibung: Dieser Task sollte nun die Kombination von AWT-Komponenten mit den *Eclipse*-eigenen SWT-Komponenten ermöglichen. Damit ist die 3D-Darstellung direkt in *Eclipse* möglich und es muss nicht unbedingt mehr ein externes Fenster geöffnet werden. Diese Möglichkeit ist aber auch weiterhin vorhanden. Diese Umsetzung ist seit der Eclipseversion 3.0M6 möglich.

geplante Zeit: Eine Stunde, da ein Prototyp bereits vorhanden war und dieser nur als Editor umgesetzt werden musste.

reale Zeit: Eine Stunde.

Event-Listener anlegen

Beschreibung: Ziel war es, einen Event-Listener für den Editor zu implementieren, damit der Editor bspw. auf Größenänderungen reagieren kann.

geplante Zeit: Geplante Zeit waren zwei Stunden.

reale Zeit: Eine Stunde reale Bearbeitungszeit. Der Editor musste nur das Interface `ControlListener` implementieren, um die gewünschten Aufgaben zu erfüllen.

16.4.4 Tests

Tests an die Framework-Architektur anpassen

Beschreibung: Auch in diesem Task waren noch restliche Aufgaben aus dem Vorgänger-Release zu erfüllen. Sämtliche Tests mussten noch an die neu geschaffene Framework-Architektur angepasst werden.

geplante Zeit: Acht Stunden wurden für die Erledigung des Tasks geschätzt, da die Bearbeitung fast alle Testklassen bzw. Testmethoden betraf.

reale Zeit: Die Dauer der Bearbeitung stimmte im Wesentlichen mit der geschätzten Zeit überein. Sie betrug sieben Stunden. Leider erwiesen sich einige Anpassungen im Laufe der Implementierung als nicht mehr möglich. Der inzwischen verwendete Milestone 8 von *Eclipse* ermöglicht das Nachladen von Klassen durch Fragmente nicht mehr. Dadurch konnten auch einige Tests nicht mehr an die Framework-Architektur angepasst werden.

16.4.5 Screenshots

Screenshots erstellen

Beschreibung: Es sollte die Möglichkeit geschaffen werden, die generierte Szene als Screenshot zu speichern.

geplante Zeit: Die geschätzte Dauer wurde mit nur einer Stunde festgehalten, da eine ähnliche Aufgabe bereits im *MuSofT-Projekt* des Lehrstuhls behandelt wurde.

reale Zeit: Die benötigte Zeit betrug eine Stunde, da nur die fertigen Codeschnipsel aus dem *MuSofT-Projekt* in den `ViewController` eingebaut und angepasst werden mussten.

Speicherdialog für Screenshots

Beschreibung: Um die erstellten Screenshots auch in beliebigen Verzeichnissen speichern zu können, war ein Speicherdialog nötig.

geplante Zeit: Eine Stunde.

reale Zeit: Zwei Stunden. Neben dem Dialog musste auch das Speichern an sich realisiert werden. Dieses war eigentlich als Teil des vorherigen Tasks geplant, ist aber mit in diesen Task eingeflossen.

16.4.6 *EFFECTS*-Perspektive

***EFFECTS*-Perspektive anlegen**

Beschreibung: Hier sollte eine eigene Perspektive für das ***EFFECTS***-Framework angelegt werden. Diese Perspektive sollte den Editor, den Package-Explorer und den ***EFFECTS***-Information-View enthalten.

geplante Zeit: Eine Stunde.

reale Zeit: Es wurden zwei Stunden gebraucht, da es einige Probleme mit dem Layout der Perspektive gab, insbesondere bei der Anordnung des `InformationView`.

Menü in der *EFFECTS*-Perspektive anlegen

Beschreibung: Neben der ***EFFECTS***-Perspektive sollte ein zusätzlicher Menüeintrag in der Menüleiste erscheinen, in dem (zunächst) 2 Einträge, Screenshot und Help, zu finden sind.

geplante Zeit: Eine Stunde.

reale Zeit: Drei Stunden. Wie auch im vorherigen Task wurde die geplante Zeit zu kurz geschätzt. Aufgrund von fehlender Erfahrung beim Anlegen eines neuen Menüs wurde mehr Einarbeitungszeit benötigt.

16.4.7 Steuerung

Tastatursteuerung überarbeiten

Beschreibung: Hier sollte die bisherige Steuerung in soweit überarbeitet werden, dass eine Kombination von Tastatur- und Maussteuerung möglich ist.

geplante Zeit: 16 Stunden wurden geschätzt. Es musste ein komplett neues Behavior implementiert werden, da die verfügbaren Java-Behaviors nicht den Anforderungen entsprachen.

reale Zeit: Null Stunden. Dieser Task wurde im Wesentlichen mit dem Task „Mausnaviagtion einfügen“ erledigt, da sich die Aufgaben weitestgehend überschnitten.

Mausnavigation einfügen

Beschreibung: Die Hauptaufgabe bestand darin, ein Mausbehavior zu implementieren. Die Navigation sollte mit Tastatur und Maus möglich sein.

geplante Zeit: 16 Stunden, da hier ebenfalls, wie auch bei der Tastatursteuerung ursprünglich angenommen, ein eigenes Behavior für die Maussteuerung notwendig war.

reale Zeit: 15 Stunden. Bei der Realisierung hat sich gezeigt, dass die beiden Tasks Tastatursteuerung und Mausnavigation durch einen einzigen Behavior zu erledigen sind. Ziel war es, das Phänomen der ungewünschten z-Achsen Rotation zu umgehen, wie dies auch beim `KeyNavigatorBehavior` der Fall ist. Diese Rotation führte dazu, dass der Horizont „schräg" zur Kamera stand, wenn erst die Kamera um x gedreht wurde, also eine Neigung der Kamera, und anschließend eine Drehung der Kamera um die y-Achse. Um dieses Problem zu lösen, ist es nötig die Matrix, welche die aktuelle Kameraposition und Drehung beinhaltet, zunächst auf die Identität zu setzen. Somit haben wir eine Rotation um 0 Grad für jede Achse. Anschließend muss erst die Rotation um die y-Achse und anschließend um die x-Achse durchgeführt werden. Hierzu werden die Winkel für diese beiden Rotationen gespeichert. Der Fehler im `KeyNavigatorBehavior` besteht also darin, dass die Matrix der Kamera schon vorherige Rotationen beinhaltet und diese einfach um einen bestimmten Winkel „weitergedreht" wird. Die Behandlung der entsprechenden Tastatur- und Mausevents stellte kein Problem dar.

16.4.8 Hilfe

Tutorial schreiben

Beschreibung: Ziel war es, ein Tutorial zu verfassen, das die Anwendung des Frameworks beschreibt.

geplante Zeit: Acht Stunden wurden hierfür geschätzt, da das Tutorial u.a. in englischer Sprache verfasst werden sollte. Auch sollte das Tutorial aus mehreren Teilen bestehen, aus beiden bestehenden Diagrammtypen, als auch aus allgemeinen Informationen zu dem Produkt.

reale Zeit: Acht Stunden wurden auch in etwa benötigt. Dabei wurde der Task in den Semesterferien verteilt auf alle PG-Mitglieder erledigt.

Extension Points beschreiben

Beschreibung: Es sollten explizit die Extension Points beschrieben und dokumentiert werden, um eine problemlose Benutzung des Frameworks sicher zu stellen.

geplante Zeit: Eine Stunde.

reale Zeit: Eine Stunde.

Einbinden der Hilfe in *Eclipse*

Beschreibung: Das geschriebene Tutorial sollte in die Hilfe-Struktur der *Eclipse*-Plattform eingebunden werden.

geplante Zeit: Eine Stunde.
reale Zeit: Zwei Stunden. Aus dem *TWIKI* mussten die HTML Seiten, die das Tutorial beinhalten, manuell extrahiert und für jedes Plugin einzeln hinzugefügt werden.

Beispieldiagramme mit allen verfügbaren graphischen Elementen
Beschreibung: Es sollten Screenshots für das Sequenzdiagramm-Plugin und für das statische Modell erstellt werden, die alle verschiedenen graphischen Elemente enthalten. Diese Bilder sollten in die Online-Hilfe eingepflegt werden.
geplante Zeit: Fünf Stunden. Der wesentliche Aufwand lag in der Erstellung einer statischen Szene, da zu dem Zeitpunkt der Implementierung noch keine Logik vorhanden war, wie graphische Elemente automatisch erzeugt bzw. geordnet dargestellt werden.
reale Zeit: Drei Stunden. Obwohl die Anordnung manuell durchgeführt werden musste, konnte eine Beispielszene relativ einfach und schnell erstellt werden.

16.4.9 Sequenzdiagramme

Grundlagen der Semantik nach Gill und Kent erarbeiten
Beschreibung: Um auf der Basis der Ausarbeitung von Gill und Kent (Gil und Kent, 1998) ein Sequenzdiagramm mit dem ***EFFECTS***-Plugin darstellen zu können, sollte eine detaillierte Ausarbeitung des Thesenpapiers erfolgen. (siehe Abschnitt 16.1)
geplante Zeit: Eine Stunde, da nur die wichtigsten und für unser Vorhaben relevanten Punkte aus dem Thesenpapier herauszuarbeiten waren.
reale Zeit: Eine Stunde wurde in etwa auch benötigt.

Graphisches Objekt Quader anlegen
Beschreibung: Dieser Task sollte ein graphisches Objekt, einen Quader, erzeugen, um die Objekte im Sequenzdiagramm visualisieren zu können.
geplante Zeit: Zwei Stunden, da das vorhandene graphische Objekt „Würfel“ aus dem statischen Diagramm übernommen werden konnte und nur an bestimmte Parameter angepasst werden musste.
reale Zeit: Zwei Stunden.

Graphisches Objekt Lebenslinie anlegen

Beschreibung: Hier sollte ein neues graphisches Objekt, die Lebenslinie eines Objektes erzeugt werden. Sie soll als dünner Zylinder visualisiert werden.
geplante Zeit: Drei Stunden, da im Gegensatz zum vorherigen Task ein neues Objekt entworfen werden musste.
reale Zeit: Drei Stunden wurden auch für die Implementierung gebraucht.

Graphisches Objekt Finalisierungskreuz anlegen
Beschreibung: Zunächst war geplant, ein graphisches Objekt „Finalisierungskreuz“ hinzuzufügen. Dieses wurde aber im Verlauf der Implementierungsphase verworfen, was zum einen an dem engen zeitlichen Rahmen und zum anderen an der Seltenheit des Auftretens des Objektes lag.
geplante Zeit: Vier Stunden.
reale Zeit: Null Stunden, da der Task nicht bearbeitet wurde.

Graphisches Objekt Aktivitätsbalken anlegen
Beschreibung: Aufgabe in diesem Task war es, eine Möglichkeit zu schaffen, die bestehende Lebenslinie um „Aktivitätszylinder“ zu erweitern.
geplante Zeit: Vier Stunden, da es im Wesentlichen darum ging, bestehende Aktivitätsbalken bzw. Lebenslinien mit „breiteren“ Aktivitätszylindern zu überdecken.
reale Zeit: Vier Stunden.

Graphisches Objekt Pfeile anlegen
Beschreibung: Auch hier sollte ein neues Objekt, ein Pfeil, implementiert werden. Dabei sollte ein Pfeil aus einem Kegel als Spitze und einem Zylinder als Pfeilkörper bestehen.
geplante Zeit: Sechs Stunden, da sowohl Ausrichtung des Pfeils auf einer Ebene, als auch die Kombination von verschiedenen graphischen Formen zu berücksichtigen war.
reale Zeit: Acht Stunden. Im Wesentlichen wurde die zeitliche Vorgabe eingehalten. Probleme gab es zunächst bei der Berechnung der Winkel der Pfeile, um verschiedene Objekte zu verbinden. Einige komplizierte mathematische Berechnungen waren nötig, um den Pfeil auf der Ebene in die entsprechende Richtung zu drehen, damit er auf das aufrufende Objekt zeigt.

Graphisches Objekt rekursiver Pfeil anlegen
Beschreibung: Hier sollte das graphische Objekt, ein Pfeil, angelegt werden, der auf das Objekt zeigt, von dem er ausgeht. Dieser Pfeil ist nötig, wenn ein Aufruf eines Objektes auf sich selbst stattfindet.

geplante Zeit: Dieser Task wurde nicht in der Planungsphase berücksichtigt und erst im Laufe der Implementierungsphase hinzugefügt.

reale Zeit: Die Bearbeitungsdauer für diesen Task betrug in etwa vier Stunden. Wie auch bei dem „normalen“ Pfeil ergaben sich wieder einige Probleme mit der Berechnung der Winkel um den Pfeil in die entsprechende Richtung zu drehen, die zunächst gelöst werden mussten.

Level of Detail für die Objekte festlegen

Beschreibung: Wie auch bei dem statischen Modell sollte es für die neuen graphischen Objekte möglich sein, bei bestimmten Entfernungen eine geringere Detailtiefe bzw. eine bestimmte Transparenzstufe zu haben.

geplante Zeit: Vier Stunden, da der Mechanismus nur noch aus dem ersten Modell an das Sequenzdiagrammm angepasst werden musste.

reale Zeit: Neun Stunden. Es musste im Nachhinein noch einiges am `ObjectCreator` und ein wenig an der Szenenberechnung geändert werden.

Darstellung der Ebenen als transparente Fläche

Beschreibung: Die als Quader dargestellten Ebenen sollen zur besseren Übersichtlichkeit transparent dargestellt werden.

geplante Zeit: Die geplante Zeit betrug vier Stunden. Auch hier konnte sich die technische Realisierung an dem Vorgängerdiagramm abgeschaut und dann an die neue Struktur angepasst werden.

reale Zeit: Der Task wurde in der geplanten Zeit von vier Stunden realisiert.

Kamerafahrt

Beschreibung: Die Aufgabe war es, eine Kamerafahrt durch eine generierte Szene zu ermöglichen. Dabei sollte das Setzen von beliebigen Wegpunkten im Editor für den Benutzer möglich sein.

geplante Zeit: Sechzehn Stunden, da die Interaktion mit dem Editor noch ein vollständig neuer Aspekt der Implementierung war.

reale Zeit: Null Stunden, da dieser Task aufgrund des engen Zeitplans nicht bearbeitet wurde.

Einarbeitung in den Abstract Syntax Tree (AST)

Beschreibung: Um ein Sequenzdiagramm zu erstellen, ist es notwendig, die Ereignisse nach einem Methodenaufruf innerhalb des Codes zu verfolgen bzw. zu registrieren. Dies sollte anhand des AST geschehen. Ziel des Tasks war es, zunächst die Grundlagen des AST zu erarbeiten.

geplante Zeit: Zehn Stunden wurden veranschlagt.

reale Zeit: Die vorgegebene Zeit von zehn Stunden wurde eingehalten. Ein wichtiges Ergebnis dieses Tasks ist es, dass ein Wechseln des *Eclipse*-Milestones problematisch ist, da es sich mit den Milestones der Version 3 noch nicht um ein offizielles Release handelt. U.a hat sich der Umgang mit der Generierung und Travesierung des AST von Milestone zu Milestone geändert. Ab dem Milestone 8 wurde zur Unterstützung ein neues Plugin genutzt (*AST-View*). Dieses Plugin kann im weitesten Sinne als Teil des JDT angesehen werden, man muss es allerdings seperat zur *Eclipse*-Umgebung hinzufügen. Dieses Plugin hat sehr zum Verständnis der AST-Struktur beigetragen.

Informationen des AST für das Sequenzdiagramm aufbereiten

Beschreibung: Die Informationen, die aus dem AST gewonnen werden konnten, mussten in einer geeigneten Form aufbereitet werden.

geplante Zeit: 30 Stunden. Es war zunächst nicht abzusehen, wie aufwendig sich die Einarbeitung in das Call-Hierarchy-Plugin gestalten würde.

reale Zeit: Die zunächst geschätzte Zeit wurde doch erheblich überschritten. Die tatsächliche benötigte Zeit betrug 45 Stunden. U.a. hat das Anpassen des Call-Hierarchy-Plugins länger gedauert. Auch sind einige unvorhergesehene Probleme aufgetreten, die im Bereich *Eclipse*, insbesondere dem Milestonewechsel, anzusiedeln sind. So sind während der Entwicklung auch noch Veränderungen am Plugin aus dem ersten Release aufgetreten, die in der implementierten Abwandlung des Plugins eingepflegt werden mussten.

Datenmodell für AST-Informationen

Beschreibung: Um die Informationen, die aus dem Quellcode gewonnen wurden, zu persistieren, sollte ein entsprechendes Datenmodell entwickelt werden.

geplante Zeit: Vier Stunden waren vorgesehen, da zunächst konzeptionelle Überlegungen eingeplant wurden.

reale Zeit: Vier Stunden. Das Modell wurde nach dem entwickelten Entwurf implementiert und die entsprechenden Tests wurden geschrieben.

Berechnung der Anzahl der Ebenen aus dem Datenmodell

Beschreibung: Dieser Task wurde erst während der Implementierung in die Taskliste aufgenommen. Es sollte die Anzahl der verschiedenen Ebenen berechnet werden, die für das Erzeugen der Szene nötig sind.

geplante Zeit: Null Stunden, da diese Taskkarte erst im Laufe der Implementierung hinzugefügt und somit nicht geschätzt wurde.

reale Zeit: Da eine etwas aufwendigere Rekursion implementiert wurde, wurde eine Zeit von etwa zwei Stunden benötigt.

Berechnung aller existierenden Objekte sowie ihre Initialposition in den Ebenen

Beschreibung: Bei diesem Task sollen die in dem Sequenzdiagramm auftretenden Objekte berechnet und ihre Position auf den Ebenen bestimmt werden.

geplante Zeit: Null Stunden, da dieser Task erst im Laufe der Implementierung hinzugefügt und somit nicht geschätzt wurde.

reale Zeit: Die Realisierung benötigte 16 Stunden, da ein größerer Aufwand darin bestand, das graphische Objekt zu ermitteln. Aus den Methodenobjekten des AST mussten die darzustellenden, graphischen Objekte berechnet werden.

Markierung der rekursiven Ebenen, Bestimmung der aufrufenden Objekte und Einfügung in die Datenstruktur

Beschreibung: Dieser Task hatte zur Aufgabe, die vom Call-Hierarchy-Plugin gelieferten Ergebnisse geeignet aufzubereiten und die benötigten Zusatzinformationen hinzuzufügen. Alle gesammelten Informationen sollten dann in der Datenstruktur gespeichert werden.

geplante Zeit: Null Stunden, da auch dieser Task während der Implementierung aufgetreten ist.

reale Zeit: 18 Stunden. Diese Aufgabe war viel aufwendiger als zunächst gedacht. Es mussten sehr viele einzelne Java-Syntax spezifische Fälle betrachtet werden. Auch musste es möglich sein, wiederverwendete Objekte auch als solche zu markieren, so dass sie nicht mehrfach im Diagramm auftauchen. Die Fallunterscheidung wurde insgesamt sehr komplex. Anzumerken ist, dass spezielle Fälle, wie innere und innere, anonyme Klassen der Einfachheit halber nicht betrachtet werden.

Berechnung des Szenegraphen aus dem Datenmodell

Beschreibung: Die in dem Datenmodell gehaltenen Daten sollten nun mittels eines Szenegraphen visuell dargestellt werden.

geplante Zeit: Null Stunden, da der Task erst in der Implementierung hinzugekommen und somit nicht geschätzt wurde.

reale Zeit: 24 Stunden wurden für die Realisierung benötigt. Es traten viele im Vorfeld nicht bedachte Kleinigkeiten auf. Weiterhin gab es Abhängigkeiten zu den beiden ebenfalls aufwendigen Tasks „Berechnung aller existierenden Objekte sowie ihre Initialposition in den Ebenen“ und „Markierung der rekursiven Ebenen, Bestimmung der aufrufenden Objekte und Einfügung in die Datenstruktur“. Diese befassen sich nämlich mit der Bereitstellung der für die Darstellung benötigten Daten.

16.4.10 Fazit

Zusammenfassend kann gesagt werden, dass die wesentlichen Aufgaben erledigt wurden. Es wurden nur die niedriger priorisierten Tasks „Kamerafahrt durch die generierte Szene“, „das graphische Objekt Finalisierungskreuz“, sowie der Schattenwurf nicht implementiert. Der höhere Zeitaufwand gegenüber der geschätzten Zeit lag im Wesentlichen daran, dass während der Implementierung noch weitere Tasks hinzu kamen, deren Zeit nicht abgeschätzt wurden. Insgesamt wurden für das Release ca. 200 Stunden benötigt, geplant waren dagegen ca. 160 Stunden.

16.5 Vorstellung der implementierten Architektur

Stephan Eisermann, Kai Gutberlet

Im Nachfolgenden soll die in diesem Release implementierte Architektur vorgestellt werden. Dazu wird zunächst die geplante Architektur beschrieben, im Anschluss daran die realisierte Architektur festgehalten und abschließend beide verglichen.

16.5.1 Beschreibung der geplanten Architektur

Die geplante Architektur dieses Releases orientierte sich an der Zielsetzung, einen neuen Diagrammtyp zu erstellen. Dieses sollte unter der Nutzung der in Release 2 realisierten Framework-Architektur geschehen.

Die Framework-Architektur bietet drei verschiedene Erweiterungspunkte (HotSpots), den Domain-HotSpot, den Diagramm-HotSpot und den Regel-HotSpot.

Domain-HotSpot Das Modell des Domain-HotSpots orientiert sich an der Aufgabenstellung, ein Sequenzdiagramm nach Gil und Kent (1998) zu erstellen. Hierzu ist es notwendig, zu einer Methode eines Objektes die Aufrufhierarchie zu speichern. Der Domain-Hotspot bietet die Möglickeit, zu einer Klasse ein Objekt, welches das `Container` Interface implementiert, zu hinterlegen. Dieser `Container` muss das Abspeichern der Aufrufhierarchie für jede Methode erlauben. Aus diesem Grund enthält der `SQContainer`, der das Interface `Container` implementiert, eine Hashmap. Diese erlaubt das Abspeichern der Aufrufhierarchie einer Methode unter dem Schlüssel `IMethodObject`. Das `IMethodObject` ist Bestandteil der *Eclipse*-internen Datenstruktur und repräsentiert eine Methode einer Klasse. Abbildung 16.1 zeigt den oben erläuterteten Sachverhalt. Die Aufrufhierarchie zu einer Methode wird dabei in einem `MethodObject` gespeichert, das wiederum eine Liste von Methodenobjekten enthält. So lässt sich zu einer Methode ein kompletter Aufrufbaum in beliebiger Tiefe abspeichern. Das `MethodObject` enthält eine Reihe von Attributen, die zur späteren Darstellung benötigt werden.

Diagramm-HotSpot Über den Diagramm-HotSpot werden die graphischen Elemente des zu realisierenden Diagramms festgelegt. Dafür ist im Framework das Interface `GraphicalObject` vorgesehen, das jedes graphische Objekt implementieren muss. Für das Sequenzdiagramm nach Gil und Kent werden folgende graphische Objekte benötigt: Pfeil, Zylinder, Qua-

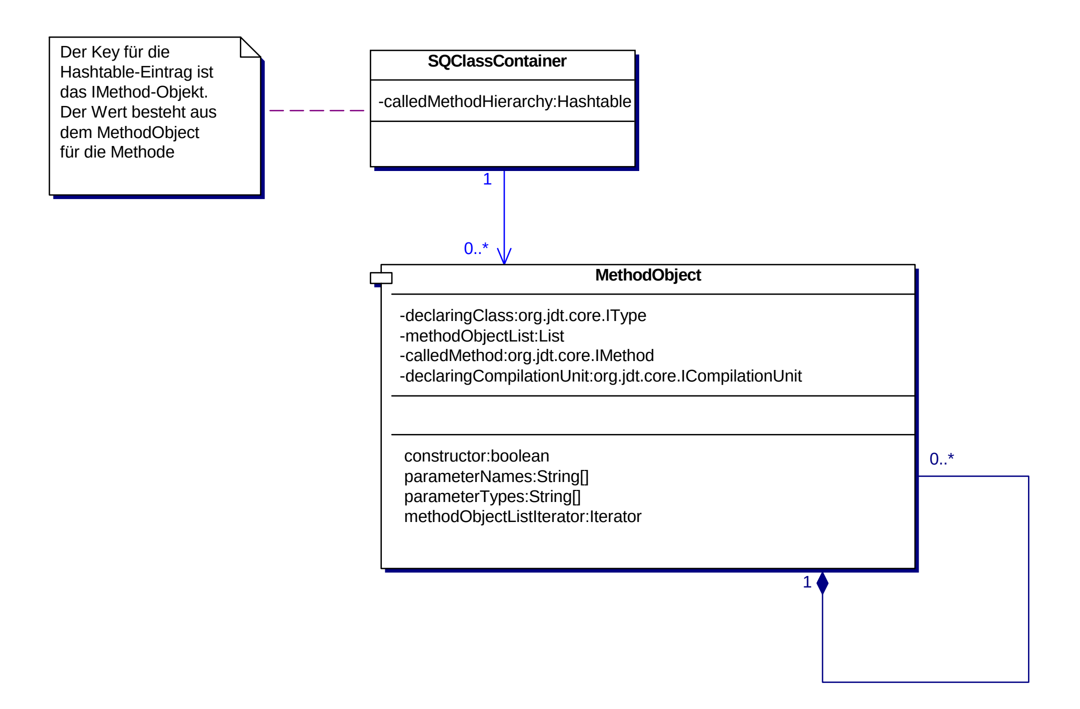


Abbildung 16.1.: Geplante Architektur des Domain-HotSpots

der, Linie, Ebene, sowie ein Pfeil, der auf sich selbst referenziert. Die Notwendigkeit, diese graphischen Objekte zu erzeugen, ergibt sich aus der in der Anforderung beschriebenen Syntax und Semantik des Sequenzdiagramms nach Gil und Kent. Diese graphischen Objekte werden in den Klassen `SQComplexArrow`, `SQComplexCylinder`, `SQComplexQuad`, `SQLifeLine`, `SQComplexLayer` und `SQComplexRecursiveArrow` definiert, die das oben angesprochene Interface `GraphicalObject` implementieren.

Regel-HotSpot Im Domain-HotSpot wurde ein Datenmodell zur Speicherung der Aufrufhierarchie einer gegebenen Methode entworfen. Dieses Datenmodell muss durch eine entsprechende Berechnungsklasse gefüllt werden. Diese Berechnungsklasse stützt sich auf ein *Eclipse*-internes Plugin, das es erlaubt, die Methodenaufrufe in einer gegebenen Methode zu ermitteln. Das Plugin wird unseren speziellen Anforderungen angepasst.

Die graphische Repräsentation der Daten erfordert zusätzliche Berechnungsklassen. Diese beziehen ihre Informationen aus dem Datenmodell, das die Aufrufhierarchie einer Methode kapselt. So sind die verwendeten Objekte aus der Aufrufhierarchie zu extrahieren. Weiterhin ist eine Metrik zur Anordnung der Objekte unter Verwendung der Aufrufhäufigkeit eines Objektes zu berechnen. Schließlich ist die Berechnung der Anzahl der Ebenen des Diagramms durchzuführen.

16.5.2 Beschreibung der realisierten Architektur

Der Schwerpunkt bei der Implementierung lag darauf, die Klassen entsprechend der Framework-Struktur zu erstellen. Im Folgenden wird daher auch die realisierte Architektur anhand der HotSpots beschrieben.

Domain-HotSpot Die in der geplanten Architektur beschriebenen Elemente `SQContainer` und `MethodObject` zur Speicherung der Aufrufhierarchie wurden umgesetzt. Während der Implementierung ergab sich die weitere Anforderung, zusätzliche Informationen zu einer gewählten Methode zu speichern. Das `MethodObject` erwies sich hierfür als nicht ausreichend, da die zusätzlich zu speichernden Informationen nur für das Wurzelelement in der Hierachie der Methodenobjekte relevant sind. Daher wurde das Datenmodell um die Klassen `MethodInformationWrapper` und `RealObject` erweitert. Der `MethodInformationWrapper` wird nun in der Hashmap der Klasse `SQClassContainer` unter dem Schlüssel `IMethodObject` gespeichert. Jeder `MethodInformationWrapper` enthält ein Methodenobjekt als Wurzel, unter dem die komplette Aufrufhierachie gespeichert wird. Zusätzlich werden alle verwendeten Objekte aus der Aufrufhierarchie in einer Hashmap gespeichert. Die Informationen eines Objekts, wie z. B. seine logische Position, werden in der Klasse `RealObject` abgelegt. Der oben erläuterte Sachverhalt wird in Abbildung 16.2 dargestellt.

Diagramm-HotSpot Die in der Anforderung beschriebenen graphischen Diagrammelemente wurden in den Klassen `SQComplexArrow`, `SQComplexCylinder`, `SQComplexQuad`, `SQLifeLine`, `SQComplexLayer` und `SQComplexRecursiveArrow` implementiert. Sie implementieren das Interface `SQGraphicalObject`, das diagrammspezifische Hilfsmethoden für die graphischen Objekte bereitstellt. Die Level-of-Detail Funktionalität der Klasse `SQComplexArrow` wurde in das Interface `SQArrowLoDVector` ausgelagert. Dieses geschah im Hinblick auf eine weitere Verwendung in späteren Releases.

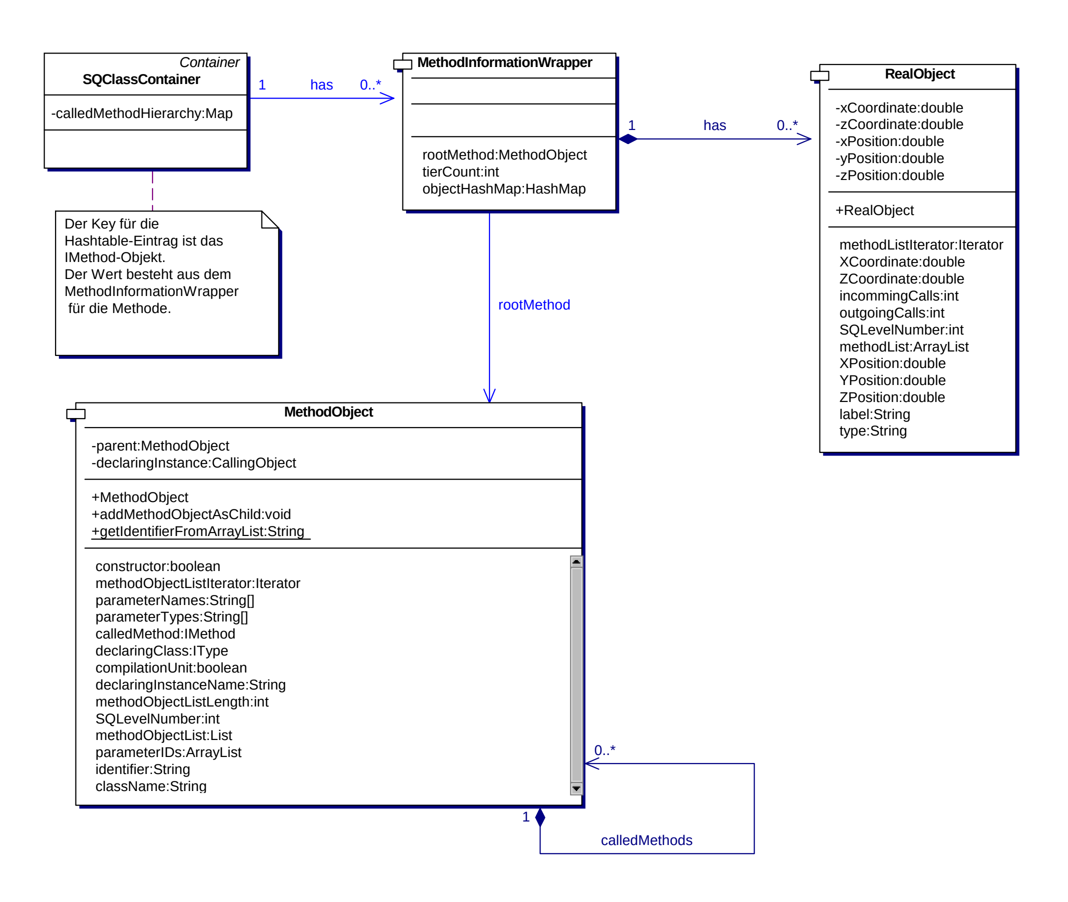


Abbildung 16.2.: Realisierte Architektur des Datenmodells

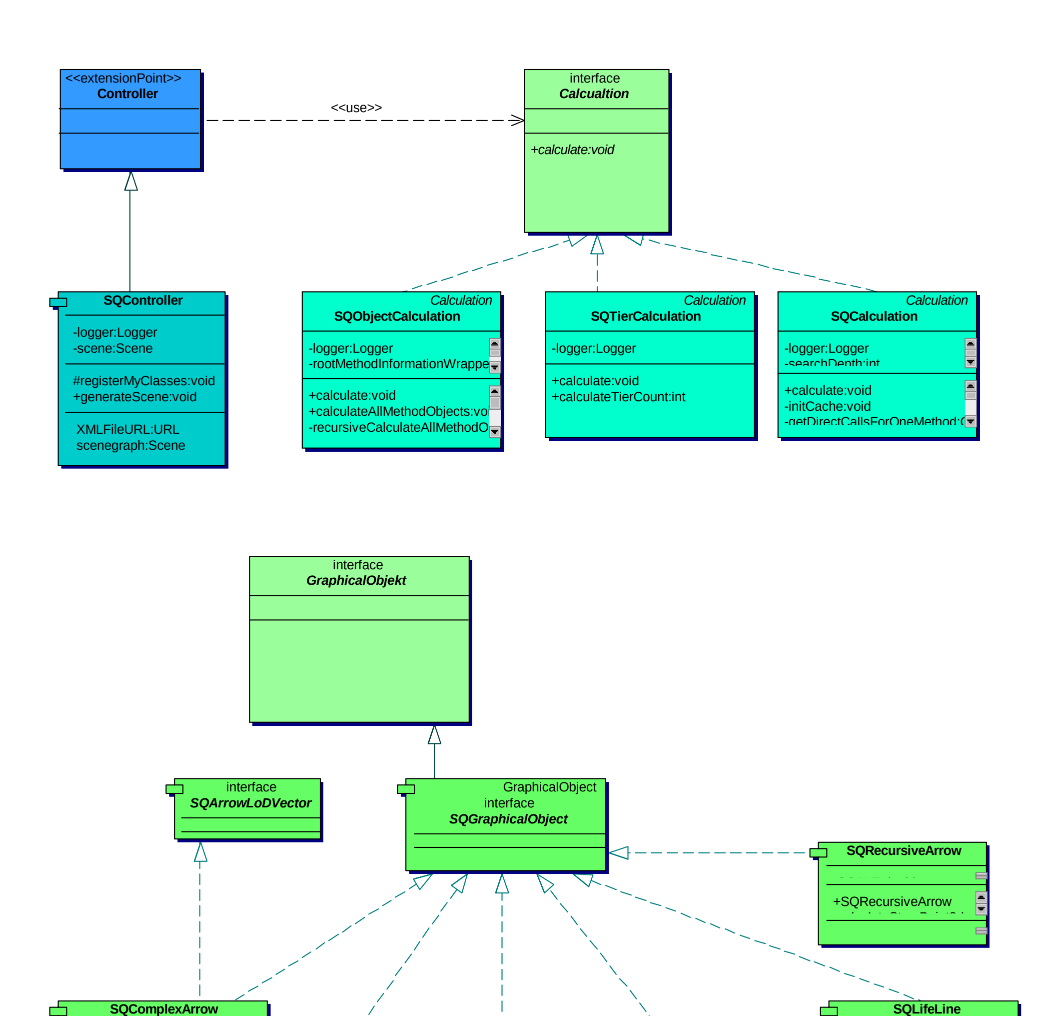


Abbildung 16.3.: Übersicht der realisierten Plugin-Struktur

Regel-HotSpot Eine Übersicht der realisierten Berechnungsklassen findet sich in Abbildung 16.3. Es wurden drei verschiedene Berechnungsklassen implementiert, nämlich `SQCalculation`, `SQTierCalculation` sowie `SQObjectCalculation`.

Die Klasse `SQCalculation` berechnet, unter der Nutzung des angepassten *Call-Hierarchie-Plugins*, die zu einer gewählten Methode gehörende Aufrufhierarchie. Diese initiale Berechnung der Aufrufhierarchie wird im Container `MethodObject` (vgl. Abbildung 16.2) gehalten, und bildet die Grundlage unseres Datenmodells.

Die Klasse `SQTierCalculation` errechnet aus der in unserem Datenmodell gespeicherten Aufrufhierarchie die Anzahl der Schichten, die später im Diagramm zu sehen sein werden. Für jede Methode wird dieser Wert als Höheninformation im jeweiligen `MethodInformationWrapper` gespeichert.

Zuletzt werden die vorhanden Objekte, ihre Verteilung auf den verschiedenen Schichten, sowie ihre logische Positionierung errechnet. Diese Funktionaliät wird von der Klasse `SQObjectCalculation` bereitgestellt. Die Informationen werden in der Klasse `MethodInformationWrapper` gehalten.

Entsprechend der Frameworkstruktur muss das diagrammspezifische Plugin in das Kernplugin eingefügt werden. Dieses geschieht durch Implementierung des vom Kernplugin zur Verfügung gestellten Extension-Point `Controller`.

16.5.3 Vergleich zwischen geplanter und realisierter Architektur

Anhand der drei HotSpots werden abschließend die geplante und die realisierte Architektur verglichen.

Domain-HotSpot Im Datenmodell ergeben sich die folgenden Unterschiede zwischen der geplanten und der realisierten Architektur. In der Hashmap der Klasse `SQClassContainer` werden nun nicht mehr Objekte vom Typ `MethodObject` sondern vom Typ `MethodInformationWrapper` gespeichert. Zusätzlich werden im `MethodInformationWrapper` noch die zugehörigen `RealObjects` verwaltet. Diese Änderungen wurden notwendig, da sich während der Implementierung weitere Anforderungen an das Datenmodell ergaben. Weiteres hierzu findet sich in der Beschreibung der realisierten Architektur.

Diagramm-HotSpot In der implementierten Architektur wurden die graphischen Elemente wie geplant realisiert. Zur besseren Wiederverwendbarkeit wurden zusätzlich die Interfaces `SQGraphicalObject` und `SQArrowLoDVector` angelegt.

Regel-HotSpot Wie in Abbildung 16.3 zu erkennen ist, wurde in der implementierten Architektur die grobe Planung der Berechnungsklassen weiter konkretisiert. Diese Verfeinerung war notwendig, da in der Modellierungsphase noch nicht alle sich aus der Implementierung ergebenden Implikationen vorherzusehen waren. So waren insbesondere weitere Kenntnisse über die Verwaltung des AST und die Architektur des zur Ermittlung von Methodenaufrufen verwendeten *Eclipse*-internen Plugins zu gewinnen.

Zusammenfassend ist zu bemerken, dass die geplante Architektur, die strukturell durch die

Verwendung des Frameworks vorgegeben war, umgesetzt worden ist. Während der Implementierungsphase erfolgten noch einige Erweiterungen im Datenmodell und die notwendige Konkretisierung der groben Planung der Berechnungsklassen.

16.6 Kunden Akzeptanztest

Christian Mocek, Jan Wessling

Im folgenden werden die Akzeptanztests, die auf den User Stories des dritten Release basieren, wie sie in Abschnitt 16.2 auf Seite 159 vorgestellt wurden.

Im *Package-Explorer* soll ein eigener Menüpunkt *EFFECTS* in das Popup-Menü eingefügt werden. In dieses Menü soll es für die Diagrammtypen möglich sein, einen Eintrag zu generieren.

Testergebnis: bestanden
Anmerkung: Visueller Test

Es sollen Lichtquellen eingebunden werden, so dass die Objekte Schatten auf den Boden werfen.

Testergebnis: nicht bestanden
Anmerkung: Dieser Test ist noch aus dem zweiten Release offen geblieben. Aufgrund der Zeitknappheit und der Entscheidung, dass Schattenwurf für das dynamische Diagramm nicht sinnvoll ist, wurde diese Anforderung wieder mit der niedrigsten Priorität bewertet.

Das Ausgabefenster soll als *Eclipse*-Editor implementiert werden.

Testergebnis: bestanden
Anmerkung: Es ist derzeit noch notwendig, eine Datei mit der Endung *.efx* manuell anzulegen um den Editor zu öffnen. Des Weiteren funktioniert dies derzeit nur unter Windows.

Sämtliche vorhandene Tests müssen auf die neue Framework-Systematik angepasst werden.

Testergebnis: nicht bestanden
Anmerkung: Problem ist hierbei der Wechsel von *Eclipse* 3.0M6 auf Version 3.0M8, bei der die Fragment Tests für die einzelnen Diagrammtypen nicht mehr funktionieren. Die Fragments versuchen Klassen nachzuladen, was allerdings von Eclipse unterbunden wird. Somit ist es rein technisch nicht mehr möglich, die Tests durchzuführen.

Es soll eine eigene *EFFECTS*-Perspektive generiert werden. In dieser Perspektive ist ein Menü hinzuzufügen.
Testergebnis: bestanden
Anmerkung: Visueller Test.

Es soll möglich sein, Screenshots von der aktuellen Ansicht zu generieren.
Testergebnis: bestanden
Anmerkung: Visueller Test: Es gibt einen entsprechenden Menüpunkt in der ***EFFECTS*** -Perspektive, über den ein Screenshot der aktuell angezeigten Szene erstell und gespeichert werden kann.

Die Steuerung soll auf Maus- und Tastatursteuerung angepasst werden.
Testergebnis: bestanden
Anmerkung: Visueller Test. Die neue Steuerung wurde auf Basis eines Ego-Shooters implementiert, also Bewegung über Pfeiltasten und Blickwinkeländerung über die Maus.

Die Extension Points müssen beschrieben werden.
Testergebnis: bestanden
Anmerkung: Sie wurden in der Hilfe beschrieben.

Es soll ein Tutorial zur Implementierung für eigene Diagrammtypen geschrieben werden.
Testergebnis: bestanden
Anmerkung: Das Tutorial ist in die Hilfe-Funktion integriert.

Die Hilfe zu den Extension Points und ein Tutorial zur Implementierung eigener Diagrammtypen soll als Online-Hilfe in *Eclipse* eingebunden werden.
Testergebnis: bestanden
Anmerkung: Visueller Test. Die Hilfe ist im *Eclipse*-Menü „Help“ über den Menüpunkt „Help Contents“ erreichbar.

Level-of-Detail (LoD): Objekte sollen ab einer bestimmten Distanz zur Kamera (optional) automatisch ausgeblendet werden können.
Testergebnis: bestanden
Anmerkung: Diese Einstellung kann nicht optional ein- bzw ausgeschaltet werden.

Auf der Zeitachse des Diagramms soll eine automatische Kamerafahrt möglich sein. Wegpunkte müssen nicht frei bestimmt werden können.

Testergebnis: nicht bestanden

Anmerkung: Diese Userstory wurde auf Grund von Zeitmangel nicht implementiert.

Für das Diagramm soll die Semantik von Gil und Kent dienen, jedoch ohne Objektzustände.

Testergebnis: bestanden

Anmerkung: Es wurde ein visueller Vergleich der Semantik von Gil und Kent mit der graphischen Ausgabe des Plugins durchgeführt.

Eine automatische Generierung des Diagramms aus vorhandenem Quellcode ist gewünscht (PG-Versammlung 26.1.04).

Testergebnis: bestanden

Anmerkung: Ein entsprechender Menüpunkt wurde in das ***EFFECTS***-Popup-Menü eingefügt. Um die Korrektheit zu testen, wurde mittels *Borland Together* das in Abbildung 16.4 dargestellte Sequenzdiagramm erzeugt und der daraus generierte Quellcode in *Eclipse* importiert. Anschließend wurde visuell der zeitliche Ablauf zwischen dem Sequenzdiagramm und der vom Plugin generierten Darstellung verglichen. Ebenso wurde der Inhalt der internen Datenstrukur mit dem von uns erwarteten Inhalt verglichen. Bei beiden Vergleichen stimmte das erhaltene Ergebnis mit dem erwartenten Ergebnis überein.

Es soll möglich sein, bei der Erzeugung eines neuen Diagramms das aktuelle Projekt als Basis zu verwenden.

Testergebnis: bestanden

Anmerkung: Da die automatische Generierung des Diagrammes anstelle der Erzeugung eines leeren Diagramms und anschließender manueller Bearbeitung gewünscht wurde, ist es somit auch möglich, aus einem Projekt eine Funktion auszuwählen und daraus das entsprechende Modell zu generieren.

Ein Objekt soll als flacher Quader angelegt werden, wobei von diesem Quader eine Lebenslinie auf der Zeitachse einzuzeichnen ist. Neue Objekte erscheinen erst auf der Zeitachse, wenn sie erzeugt werden.

Testergebnis: bestanden

Anmerkung: Visueller Test.

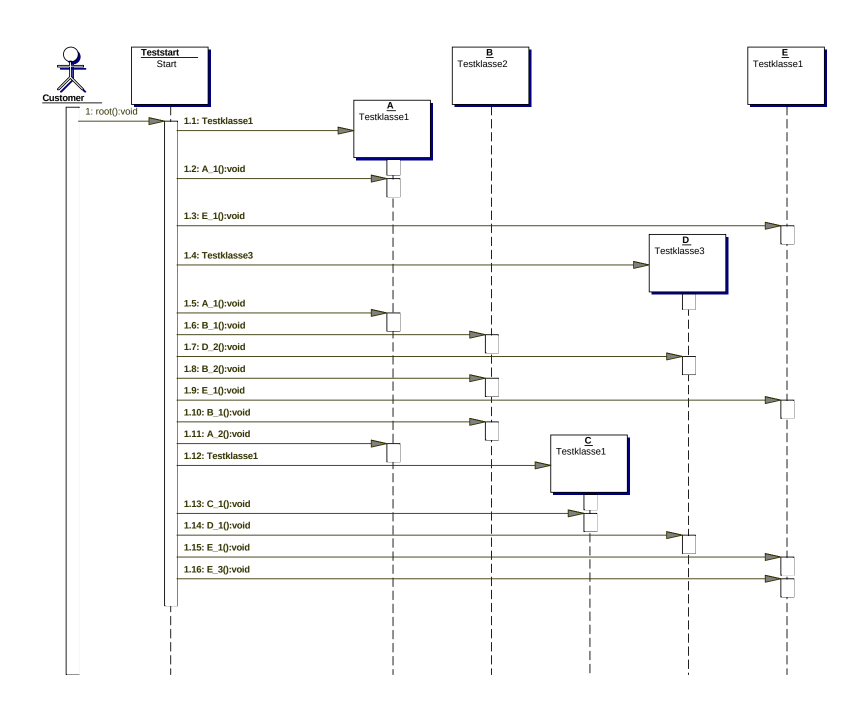


Abbildung 16.4.: Sequenzdiagramm für den Kundentest

Die Aktivitätsbalken sollen als Röhre auf der Lebenslinie dargestellt werden.
Testergebnis: bestanden
Anmerkung: Visueller Test.

Pro Methodenaufruf ist eine Ebene erforderlich, auf der der Methodenaufruf durch einen Pfeil dargestellt wird.
Testergebnis: bestanden
Anmerkung: Aufgrund der Übersichtlichkeit wurde entschieden, dass eine visuelle Ebene nur dann eingezeichnet wird, wenn ein neues Objekt erzeugt wird.

Bei einem Methodenaufruf ist neben dem Pfeil für den Methodenaufruf auch noch der Parameteraufruf mittels eines Pfeiles einzuzeichnen.
Testergebnis: nicht bestanden
Anmerkung: Aus Zeitgründen wurde diese User Story nicht implementiert.

Die beiden Arten für Methoden- und Parameterpfeile sollen unterschiedliche Farben besitzen.
Testergebnis: nicht bestanden
Anmerkung: Da die Parameterpfeile nicht implementiert wurden, wurde dieser Test nicht bestanden. Für zukünftige Implementierungen ist dies allerdings kein Problem, da die Klasse, die den Pfeil repräsentiert, eine Funktion zur Änderung der Farbe zur Verfügung stellt.

KAPITEL 17

Beschreibung des vierten Release

17.1 Einleitung

Armin Bruckhoff

Das vierte Release bestand zunächst darin, ein neues Diagramm auf Basis des Klassenpaketdiagramms von Engelen (Engelen, 2000) zu implementieren und in diesem Editierbarkeit, Codesynchronisation und Interaktion zur Verfügung zu stellen. Im darauf folgenden fünften Release der ursprünglichen Planung sollten dann die Möglichkeiten zur Interaktion aus dem speziellen Diagramm in das Framework migriert werden.

Beim genaueren Ausarbeiten des vierten Release, insbesondere der User Stories, stellte sich dann aber heraus, dass die Implementierung des neuen Diagramms und Einbindung der Interaktion zu umfangreich für einen Releasezyklus von fünf Wochen sind. Darüberhinaus muss die Verschiebung der Interaktion in das Framework schon direkt bei der Konzeptionierung im vierten Release beachtet werden. Daher kann sie direkt im Framework umgesetzt werden. Das fünfte Release der ursprünglichen Planung kann somit entfallen, und die dadurch eingesparte Zeit für das vierte Release aufgewandt werden.

Daraus ergab sich folgende Neuaufteilung der ausstehenden Aufgaben von Release 4: In Release 4*a* wird zunächst das Klassenpaketdiagramm und grundlegende Interaktion (Auswählen, Verschieben von Objekten, etc.) implementiert. In Release 4*b* kommen dann Codesynchronisation und Syntaxprüfung hinzu. Dadurch wird es möglich, sowohl das Diagramm in Quellcode zu überführen als auch Quellcode zu analysieren und in einem Diagramm darzustellen. Release 4*a* und 4*b* haben beide am jeweiligen Ende eine Abgabe mit Abnahmetest der Kundengruppen.

Diese Neuaufteilung bewirkt allerdings einen Bruch mit dem XP–Prozess. Es gibt nun keine klare Trennung der beiden Releases. Das Ende von Release 4*a* markiert lediglich einen Zwischenstand der Gesamtentwicklung des ganzen Release 4, die erst am Ende von Release 4*b* abgeschlossen ist.

17.2 User Stories

André Kupetz, Michael Nöthe

Da, wie Eingangs erwähnt, zwei offizielle Releaseabgaben stattfanden, sind auch die User Stories in zwei Hälften aufgeteilt.

Die im Folgenden aufgeführten User Stories stellen die Anforderungsdefinition für die Releases 4*a* und 4*b* des *Eclipse*-Plugins dar. Das Gesamtziel der beiden Releases war es, das Plugin dahingehend zu entwickeln, dass es als graphischer Editor zur dreidimenionalen Gestaltung von Softwarestrukturen genutzt werden kann.

User Stories des Release 4*a*

Interaktion

Objekte sollen markiert werden können (auch mehrfach), um sie anschließend verschieben, löschen oder ändern zu können. Neue Objekte sollen eingefügt werden können.

Neue Syntaxelemente

Es sollen

- Information Cubes als halbtransparente Würfel zur Paketvisualisierung,
- Cone Trees als Visualisierung der Vererbungsbeziehungen und
- Interfaces als Kugeln dargestellt werden (Klassen werden weiterhin als Würfel visualisiert)

Assoziationen

Assoziationen sollen durch Pipes dargestellt werden (Kardinalitäten als Text, verschiedene Farben für Assoziation, Erbung, Implementierung). Assoziationen zwischen Paketen bzw. zwischen in ihnen enthaltenen Klassen sollen durch genau eine große Pipe zwischen den Paketen, eine sogenannte Sammelpipe, dargestellt werden.

Menüleiste und Toolbar

Zum Auswählen der Werkzeuge soll es eine Menüleiste und eine Toolbar geben.

Automatische Anpassung des Layouts

Es soll möglich sein, Objekte auf den Cone Tree zu ziehen, so dass sich diese in die Erbungshierarchie einfügen. Des Weiteren soll bei dem Einfügen einer zusätzlichen Erbungsrelation zu einer Klasse der entsprechende Cone Tree neu angeordnet werden.

SONDERFALL: Erbung über Paketgrenzen hinweg

Klassen, die an Erbung über Paketgrenzen hinweg beteiligt sind, werden in jedem der betroffenen Pakete dargestellt. In ihren „Nicht-Heimat-Paketen" werden diese Klassen gesondert dargestellt, um anzuzeigen, dass sie nicht in dieses Paket gehören. Es handelt sich um sogenannte Proxies.

Einfügen von Assoziationen

Assoziationen sollen in das Diagramm eingefügt werden können. Start- und Zielpunkt sollen jeweils durch Anklicken gewählt werden. Danach soll das Diagramm direkt aktualisiert werden.

Schatten

Die graphischen Objekte sollen Schatten auf den Boden des Universums projizieren.

User Stories des Release $4b$

Persistenz des graphischen Layouts

Nachdem ein automatisch erstelltes Diagramm vom Benutzer verändert wurde, soll sichergestellt sein, dass es nach Schließen des Diagramms und/oder Eclipse genau so wieder angezeigt werden kann wie vor dem Schließen.

Neuanlegen von Elementen

Das Neuanlegen von Elementen soll sowohl durch graphisches Einfügen, als auch mit Hilfe eines Dialogs möglich sein.

Autolayout

- Vererbungsbeziehungen sollen als Cone Trees dargestellt werden.
- Die Größe des Pakets soll sich entsprechend der enthaltenen Klassen und Unterpakete ändern.
- Die Größe des Universums soll sich an die enthaltenen Elemente anpassen.
- Klassen, Interfaces und Pakete sind gemäß der Pakethierarchie geschachtelt.

Syntaxprüfung

Angelehnt an die bisherigen XML-Dateien zur Diagrammkonfiguration sollen verschiedene Klassen zur Prüfung der Syntax angegeben werden können. Es sollen folgende Regeln überprüft werden:

- Keine Mehrfacherbung (außer bei Interfaces).
- Alle Kanten haben zwei Endpunkte.
- Objekte überlappen sich nicht.
- Information Cubes können geschachtelt werden.
- Erbung auf Cone Trees.

Synchronisation

Code und Diagramm sollen synchron sein.

Inspektor (*Propertybox*)

Es soll eine Propertybox erstellt werden, mit dessen Hilfe das Ändern und Anzeigen der Kardinalitäten, Klassennamen, Farben etc. komfortabel ermöglicht wird.

Proxies

Für die Erbung über Paketgrenzen hinaus sollen Proxies eingeführt werden.

Graphische Feinheiten und Bedienbarkeit

- Die vordere Wand des Information Cubes soll ausgeblendet werden.
- Der Detailgrad der angezeigten Informationen soll wählbar sein (Ausblenden der Beschriftungen etc.)
- Es sollen Schatten anstatt der Fadenkreuze angezeigt werden.
- Der Mauszeiger soll sich abhängig vom aktuellen Bearbeitungsmodus ändern.
- Die Menüleiste soll in *Eclipse* eingebunden werden.

Assoziationen

Assoziationen zwischen Paketen sollen als Sammelpipes zwischen den Paketen dargestellt werden. Weiter soll es möglich sein, verschiedene Arten von Assoziationen zu benutzen, z.B. gerichtete und ungerichtete.

17.3 Systemmetapher

Jan Wessling, Armin Bruckhoff

Die Systemmetapher des zusammengefassten vierten und fünften Releases ist eine Realisierung des Model–View–Controller Konzepts. In diesem Release wurde die frühere Umsetzung analysiert und eine striktere Trennung der einzelnen Teile implementiert: Bisher war es so, dass Teile der Kommunikation direkt zwischen Model und View stattfanden. Dies wurde unterbunden. Die Kommunikation von Model und View erfolgt ausschließlich über den Controller.

Der Controller wurde dabei als Zustandsautomat realisiert, der die verschiedenen Bearbeitungsmodi in seinen Zuständen erfasst. Alle Eingaben vom Benutzer werden über den Automaten an das Datenmodell weitergereicht. Desweiteren sind keine Referenzen im Datenmodell auf Teile des Views vorhanden. Gleichzeitig wurde der Zugriff auf das Datenmodell mit Hilfe einer Fassade gekapselt, um den Zugriff auf unsere Container zu vereinfachen.

17.4 Reflexion über die Tasks

Semih Sevinç, Michael Nöthe, Michael Pflug, René Schönlein

Im Nachfolgenden wird über die Tasks des vierten und fünften Releases reflektiert. Hierzu werden die einzelnen Tasks kurz vorgestellt und insbesondere auf die Differenzen in den Zeitabschätzungen eingegangen.

17.4.1 Taskreflexion Release $4a$

Neue Syntaxelemente

Information Cubes

Beschreibung:	Pakete sollten als durchsichtige Information Cubes dargestellt werden.
geplante Zeit:	2 Stunden
reale Zeit:	Dieser Task wurde in 1,5 Stunden erledigt, da die Cubes aus dem ersten Release übernommen werden konnten.

Ausblenden der Vorderseite von Information Cubes

Beschreibung:	Bei der Darstellung eines Paketes sollten die dem Betrachter zugewandten Wände ausgeblendet sein.
geplante Zeit:	8 Stunden
reale Zeit:	Dieser Task wurde aufgrund der niedrigen Priorität und des Zeitmangel nicht bearbeitet.

Cone Trees (ohne drehbare Teilbäume)

Beschreibung:	Die Erbungshierarchie von Klassen sollte mit Cone Trees dargestellt werden.
geplante Zeit:	12 Stunden
reale Zeit:	10 Stunden wurden für diesen Task benötigt. Für die Berechnung der Anordung der Klassen auf dem Cone wurde der Algorithmus von Carriere and Kazman (LITREF EINBAUEN) in einer Implementierung des Diplomanden Ingo Röpling verwendet.

Klassenwürfel aus dem Staticmodel übernehmen

Beschreibung:	Die Klasssenwürfel aus dem ersten Release sollten bei diesem Task übernommen und ggf. angepasst werden.
geplante Zeit:	1 Stunde
reale Zeit:	1 Stunde

Virtuelle Klassen (Proxies) sollten nicht auswählbar sein

Beschreibung:	Bei Erbung über Paketgrenzen hinaus sollten sogenannte Proxies benutzt werden, um die Klassen, die nicht in dem jeweiligen Paket sind, dort anzeigen zu können. Diese virtuellen Klassen sollen aber nicht selektierbar sein.
geplante Zeit:	2 Stunden
reale Zeit:	Aus Zeitgründen wurde dieser Task nicht bearbeitet.

Interfaces

Beschreibung:	Interfaces sollten als Kugel dargestellt werden.
geplante Zeit:	2 Stunden
reale Zeit:	2 Stunden

Assoziationen

Assoziationen

Beschreibung:	Skalierbare Pfeile sollten die Assoziationen zwischen Elementen darstellen.
geplante Zeit:	8 Stunden.
reale Zeit:	Für diesen Task wurden 60 Stunden benötigt. Es gab hier sehr viele Probleme mit der Rotation der Pfeile in beide Richtungen. Letztendlich wurde eine Methode aus dem EasyViewer übernommen.

Kardinalitäten

Beschreibung:	Die Kardinalitäten für Assoziationen zwischen zwei Elementen sollte dargestellt werden.
geplante Zeit:	1 Stunde
reale Zeit:	Da dieser Task die Priorität drei hatte und nicht mehr genug Zeit vorhanden war, wurde er nicht bearbeitet.

Toolbar erstellen

Menüleiste und Toolbar erstellen

Beschreibung:	Es sollte eine Menüleiste und eine Toolbar zum Auswählen der Werkzeuge erstellt werden.
geplante Zeit:	1 Stunde
reale Zeit:	Benötigt wurden 4 Stunden. Die Klasse `JToolBar` musste gekapselt werden um ein einzelnes, kontrolliertes Ereignis erzeugen zu können. Die Ereignis- und Listenerverwaltung musste ebenfalls implementiert werden.

Extension Point für die Toolbar definieren

Beschreibung:	Die Toolbar sollte erweiterbar gestaltet werden, damit später weitere Buttons leichter angefügt werden können.
geplante Zeit:	3 Stunden wurden geplant.
reale Zeit:	Dieser Task wurde aus Zeitgünden nicht bearbeitet.

Interaktion

Objekte auswählen

Beschreibung:	Objekte sollten auswählbar sein. Es sollte auch eine mehrfache Auswahl möglich sein.
geplante Zeit:	3 Stunden
reale Zeit:	Benötigt wurden 6 Stunden, da die Umsetzung des Pickings komplexer als zunächst erwartet war.

Ausgewählte Objekte graphisch hervorheben

Beschreibung:	Die Auswahl von Objekten sollte graphisch dargestellt werden.
geplante Zeit:	4 Stunden
reale Zeit:	Dieser Task wurde in 2 Stunden abgeschlossen, da lediglich die vorhandene *Bounding-Box* des jeweiligen Elementes gezeichnet werden musste.

Objekte verschieben

Beschreibung:	Ausgewählte Objekte sollten verschiebbar sein.
geplante Zeit:	4 Stunden
reale Zeit:	Benötigt wurden für diesen Task über 25 Stunden. Es gab zunächst erhebliche Probleme, die Translationen mit der Maus durchzuführen, da diese ein zweidimensionales Eingabemedium ist und diese Eingabe auf einen dreidimensionalen Richtungsvektor abgebildet werden muss. Aufgrund der eigentlich trivialen Lösung kann man hier von einem typischen Fall ausgehen, wo mangelnde Erfahrung im Umgang mit Java3D und Matrizen das Hauptproblem waren. Die Lösung bestand darin, den Start und Endpunkt auf dem Canvas in einen Punkt im Raum umzurechnen und aus der Differenz der beiden den Richtungsvektor zu erhalten. Dieser wird dann entsprechend skaliert, je nachdem, wie weit die Objekte entfernt sind. Dies führte zu einem weiteren, recht zeitaufwändigen Teilproblem, nämlich der Berechnung der Distanz eines Objektes zur Ebene, die von der x- und y-Achse der Kamera aufgespannt wird. Es wird jetzt hierzu der Normalenvektor der Ebene berechnet und im `CameraChangedListener` mit übergeben. Der `ObjectTranslator` berechnet den Abstand des Objekts zur Ebene.

Objekte löschen

Beschreibung:	Ausgewählte Objekte sollten gelöscht werden können.
geplante Zeit:	1 Stunde
reale Zeit:	1 Stunde

Objekte anlegen

Beschreibung: Es sollte möglich sein neue Objekte anzulegen.
geplante Zeit: 3 Stunden
reale Zeit: Dieser Task ist im Task „Zustandsautomat realisieren“ enthalten.

Neue Objekte automatisch benennen

Beschreibung: Neu angelegte Objekte sollten einen eindeutigen Namen bekommen.
geplante Zeit: 1 Stunde.
reale Zeit: Dieser Task wurde im Task „Aktualisieren der Szene“ behandelt.

Schattenwurf

Alle Elemente sollten einen Schatten auf den Boden projizieren

Beschreibung: Es sollte ein Konzept für den Schattenwurf erstellt werden.
geplante Zeit: 2 Stunden
reale Zeit: Benötigt wurde 1 Stunde. Folgendes Ergebnis wurde erarbeitet: Java3D unterstützt keinen Schattenwurf. Die einzige Möglichkeit hierfür ist es, einen Richtungsvektor als „Lichtquelle“ zu definieren und dann die Schatten als Polygon zu berechnen und dieses flach auf den Boden zu legen. Da der Aufwand definitiv zu hoch erschien, wurde diese User Story nicht weiter verfolgt.

Weitere Tasks

Im Folgenden werden die Tasks aufgelistet, die am Anfang dieses Releases nicht eingeplant, aber für die Umsetzung vieler User Stories dringend notwendig waren.

Controllerkonzept erstellen

Beschreibung: Das Plugin sollte nach dem MVC-Konzept gestalltet werden. Der Controller sollte einen Zustandsautomaten benutzen, um die verschiedenen Interaktionsmodi zu unterstützen.
geplante Zeit: 16 Stunden
reale Zeit: 16 Stunden

Datenmodell entwickeln

Beschreibung: Ein Datenmodell musste erstellt werden. In diesem Release musste ein umfangreicheres Modell erstellt werden, da neben Klassen, Interfaces und Paketen nun auch Beziehungen zwischen diesen Elementen verwaltet werden sollten.
geplante Zeit: 8 Stunden

reale Zeit: 8 Stunden

Datenmodell implementieren

Beschreibung: Das im vorangehenden Task entwickelte Datenmodell musste implementiert werden.
geplante Zeit: 5 Stunden
reale Zeit: Es wurden 8 Stunden benötigt, da das Datenmodell während der Implementierung weiterentwickelt wurde und die Implementierung dementsprechend angepasst werden musste.

Zustandsautomaten entwerfen

Beschreibung: Der Zustandsautomat sollte die verschiedenen Modi, wie z.B. Kamera- oder Verschiebemodus, verwalten und erkennen können, wann die dargestellte Szene gespeichert werden muss.
geplante Zeit: 1 Stunde
reale Zeit: Benötigt wurden 4 Stunden. Der Automat wurde nach dem Artikel (Yacoub und Ammar, 1998) modelliert.

Zustandsautomaten realisieren

Beschreibung: Der im vorherigen Task entwickelte Automat musste implementiert werden.
geplante Zeit: 16 Stunden
reale Zeit: Es wurden 20 Stunden benötigt. Die Abschätzung erwies sich als schwierig, da angenommen wurde, dass die einzelnen Zustände einer ständigen Weiterentwicklung unterliegen würden.

Pluginstruktur anlegen

Beschreibung: Die Datei `fragment.xml` musste um den Eintrag für den Menüpunkt zum Erzeugen des Diagramms erweitert werden.
geplante Zeit: 1 Stunde
reale Zeit: 1 Stunde

Auslesen des Datenmodells und Aktualisierung der Szene

Beschreibung: Das Datenmodell sollte bei Bedarf ausgelesen werden können, um mit dessen Informationen die darzustellende Szene neu aufbauen zu können.
geplante Zeit: 20 Stunden

reale Zeit: In 14 Stunden konnte dieser Task abgeschlossen werden. Die Zeitabschätzung beruhte auf der Annahme, dass beim Auslesen noch Berechnungen durchgeführt werden müssen. Zum Update der Szene muss lediglich der alte Inhalt gelöscht und neu aus dem Datenmodell ausgelesen werden. Die Berechnung findet schon vor dem Schreiben ins Datenmodell statt.

17.4.2 Taskreflexion Release *4b*

Persistenz des graphischen Layouts

Trennung von Model und View: Referenzen auf 3D-Objekte aus Model entfernen.

Beschreibung: Dieser Task wird für die Speicherung benötigt, da die Klassen der 3D-Objekte nicht das Interface `Serializable` implementieren. Außerdem kommt die Struktur des Plugins so dem MVC-Konzept näher.

geplante Zeit: 5 Stunden

reale Zeit: 7 Stunden

Trennung von Model und View: Kontrollmethoden aus `DataModelHelper` in den `PluginController` verschieben

Beschreibung: Diese Kapselung war notwendig, um das Speichern zu realisieren.

geplante Zeit: 3 Stunden

reale Zeit: 3 Stunden

Dirty-Flag im Datenmodell

Beschreibung: Dieses Flag sollte zur Erkennung von geänderten Objekten dienen, die dann im View neu dargestellt werden müssen.

geplante Zeit: 5 Stunden

reale Zeit: 5 Stunden

Überarbeiten der Container-Strukturen

Beschreibung: Das Interface `Container` wurde in eine abstrakte Klasse umgewandelt, um allgemeine Funktionalität bereits an dieser Stelle den spezialisierten Containern zur Verfügung zu stellen.

geplante Zeit: 4 Stunden

reale Zeit: 5 Stunden

Erweiterung der Zugriffsmethoden auf den Container

Beschreibung: Es gibt zu jeder Resource einen `MainContainer` und potentiell eine Liste mit weiteren Containern, z.B. für Assoziationen an einer Klasse. Daher mussten Methoden geschrieben werden, die diese Container verwalten.

geplante Zeit: 2 Stunden

reale Zeit: 1 Stunde

Verweis auf `IJavaElement` im Container intern nur noch als String speichern und Zugriffsmethoden bereitstellen

Beschreibung: Dieser Task war ebenfalls notwendig, um das Speichern zu ermöglichen. Das `IJavaElement` sollte nicht als Objekt gespeichert werden, sondern nur ein Verweis darauf.

geplante Zeit: 3 Stunden

reale Zeit: 3 Stunden

Verwaltung einer internen Liste aller benutzten Container im `PluginController`

Beschreibung: Diese Aufgabe sollte das Speichern erleichtern. Es war mit Hilfe dieser Liste besonders einfach alle Container zu speichern, da nicht erst das gesamte Datenmodell traversiert werden musste.

geplante Zeit: 1 Stunde

reale Zeit: 1 Stunde

Flag im Container, ob es sich um einen sogenannten `MainContainer` handelt, d.h. ob er direkt an einer Resource hängt oder nicht

Beschreibung: Es gibt zu jedem Element einen `MainContainer` und potentiell eine Liste mit weiteren Containern, z.B. für Assoziationen an einer Klasse.

geplante Zeit: 1 Stunde

reale Zeit: 1 Stunde

Container dem *JavaBean*-Standard anpassen

Beschreibung: Die Container mussten, zur Serialisierung in eine *.efx-Datei, dem *JavaBean*-Standard angepasst werden. Das bedeutet, dass für alle Attribute Get- und Set-Methoden eingefügt werden mussten.

geplante Zeit: 3 Stunden

reale Zeit: Benötigt wurden 4 Stunden. Es stellte sich heraus, dass die Klassen von Java3D nicht dem *JavaBean*-Standard entsprechen. Es wurde eine eigene *JavaBean*-konforme Klasse zur Speicherung von Tripeln angelegt.

Speicherung des Datenmodells in einer efx-Datei

Beschreibung:	Dieser Task befasst sich mit der eigentlichen Serialisierung in die efx-Datei. Das Format der Datei ist XML.
geplante Zeit:	6 Stunden
reale Zeit:	6 Stunden

Laden des Datenmodells aus einer efx-Datei

Beschreibung:	Die Szene soll anhand der Objekte und deren Positionsinformationen in der efx-Datei wiederhergestellt werden können.
geplante Zeit:	8 Stunden
reale Zeit:	6 Stunden

Neuanlegen von Elementen sowohl durch graphisches Einfügen als auch mit Hilfe eines Dialogs.

Einfügen durch den Wizard, den *Eclipse* bereitstellt, und Synchronisation

Beschreibung:	Das eingefügte Element musste in das Datenmodell aufgenommen und von dort in die Szene übertragen werden.
geplante Zeit:	4 Stunden
reale Zeit:	Dieser Task wurde an dieser Stelle nicht bearbeitet. Wie sich herausstellte, deckte der Task „Codesynchronisation“ (s. 17.4.2) diesen Fall bereits ab.

Einfügen von Elementen an die aktuelle Kameraposition.

Beschreibung:	Es sollte möglich sein, Elemente vor der Kameraposition in die Szene zu legen.
geplante Zeit:	6 Stunden
reale Zeit:	6 Stunden

Autolayout

Konzeptionierung der Anordnung

Beschreibung:	Beim Generieren eines Diagramms aus dem Quellcode mußte eine Anordnung der Elemente entworfen werden. Dazu gehörte die Anordnung von Cone Trees, Paketen, Unterpaketen, sowie einzelner Klassen und Interfaces, die in keiner Erbungsbeziehung stehen.
geplante Zeit:	8 Stunden
reale Zeit:	8 Stunden

Klassen, Interfaces, Cone Trees und Unterpakete innerhalb eines Paketes anordnen

Beschreibung: Die im vorherigen Task gewonnenen Erkenntnisse mussten jetzt umgesetzt werden. Dies geschah mit einem rekursiven Algorithmus, der zunächst Unterpakete anordnet und dann das Paket, in dem sie enthalten sind. Ebenso mussten Cone Trees entsprechend ihrer Größe im Raum angeordnet werden.

geplante Zeit: 35 Stunden. Dieser Task wurde als sehr komplex eingeschätzt. Es mussten Unterscheidungen berücksichtigt werden, um die Anordnung angemessen zu gestalten. Z.B. war zu unterscheiden, ob alle Objekte mit einem Mal oder nur einzelne Objekte anzuordnen sind.

reale Zeit: 34 Stunden. Leider konnten aus Zeitgründen nicht alle Sonderfälle berücksichtigt werden, wie z.B. die Vererbung über Paketgrenzen hinaus. Außerdem ist der implementierte Algorithmus noch nicht optimal. Die Paketgrößen etwa werden unter bestimmten Umständen zu groß berechnet.

Cone Trees anordnen

Beschreibung: In diesem Task ging es um den Aufbau der Cone Trees, also um die Anordnung der einzelnen Klassen auf den Cones entsprechend ihrer Erbungsbeziehungen.

geplante Zeit: 5 Stunden. Der Algorithmus des EasyViewer Projektes sollte benutzt werden.

reale Zeit: 10 Stunden. Der Algorithmus war nicht für die interaktive Anordnung von Cone Trees ausgelegt und musste entsprechend angepasst werden.

Toplevel-Pakete und Klassen / Interfaces im `DefaultPackage` im Universum anordnen

Beschreibung: Hier ging es darum, aus dem Code zu erkennen, ob eine Klasse oder ein Interface zum `Defaultpackage` gehört oder nicht.

geplante Zeit: 10 Stunden. Es traten im Vorfeld bereits Probleme mit der Paketdeklaration auf.

reale Zeit: 2 Stunden. Die zuvor befürchteten Probleme sind nicht aufgetreten.

Struktur zum Generieren eines Diagramms anlegen

Beschreibung: Dieser Task beschäftigte sich mit dem Anstoßen des Layoutalgorithmus.

geplante Zeit: 4 Stunden

reale Zeit: 4 Stunden

Syntaxprüfung

Konzeptionelle Ausarbeitung des Syntaxchecks auf graphischer Ebene und dem Datenmodell

Beschreibung: Hier gab es eine Unterteilung in zwei Bereiche. Der erste Bereich befasst sich mit dem graphischen Syntaxcheck mit Hilfe einer Kollisionsabfrage. Es sollte geprüft werden können, ob sich z.B. eine Klasse innerhalb eines Paketes befindet. Der zweite Bereich beinhaltet eine Syntaxprüfung auf Codeebene, die z.B. für Erbungsbeziehungen zuständig ist.

geplante Zeit: 10 Stunden

reale Zeit: 10 Stunden

Kollisionsabfrage realisieren

Beschreibung: Die Kollisionsabfrage ist eine Voraussetzung für die graphische Syntaxprüfung.

geplante Zeit: 40 Stunden. Die Benutzung der *Octree*-Datenstruktur aus Performanzgründen wurde angestrebt. Die *Octree*-Datenstruktur ermöglicht eine Unterteilung des dreidimensionalen Raums in hierarchische Segmente, um die Lage von Objekten zu ihren Nachbarn schneller abschätzen zu können.

reale Zeit: Abgebrochen nach 20 Stunden. Die *Octree*-Datenstruktur erwies sich in der Kürze der Zeit als nicht beherrschbar. Selbst die vereinfachte Kollisionsabfrage mit Hilfe von *Bounding-Box* konnte in der verbleibenden Zeit nicht mehr implementiert werden. Als Problem stellte sich dabei der Umfang der *Bounding-Box* heraus. Dieser war so groß, dass selbst wenn die Objekte nur dicht nebeneinander lagen, aufgrund der *Bounding-Boxen* bereits eine Überschneidung ausgegeben wurde.

Syntaxcheck auf graphischer Ebene realisieren

Beschreibung: Dieser Task sollte mit Hilfe der Kollisionsabfrage verschiedene Regeln graphisch überprüfen, z.B. die Überschneidungsfreiheit von Klassenwürfeln und Interfacekugeln.

geplante Zeit: 20 Stunden

reale Zeit: Abgeborchen nach 8 Stunden. Wie schon erwähnt, war dieser Task existentiell abhängig von der Kollisionsabfrage, die nicht mehr implementiert wurde, daher wurde der Task nach der Hälfte der Zeit eingestellt.

Syntaxcheck auf AST-Ebene realisieren

Beschreibung: Im Datenmodell sollte überprüft werden, ob Mehrfacherbung oder zyklische Vererbung vorliegt. Weiterhin sollte überprüft werden, dass nur erlaubte Beziehungen zwischen den Objekten angelegt werden dürfen, wie z.B. Erbungbeziehungen zwischen Klassen und nicht zwischen Paketen.
geplante Zeit: 10 Stunden
reale Zeit: 18 Stunden. Es waren besonders viele Fallunterscheidungen zu berücksichtigen. Des Weiteren konnten aus zeitlichen Gründen nicht alle Sonderfälle, z.B. innere Klassen, berücksichtigt werden.

Code- und Diagrammsynchronismus

Erstellen eines Controllers für den Datenaustausch zwischen Datenmodell und AST
Beschreibung: Aufgabe dieses Tasks war es hauptsächlich, die jeweils benötigten Informationen aus den passenden Datenstrukturen, d.h. entweder dem JDT-Datenmodell bzw. dem AST oder dem ***EFFECTS***-Datenmodell, auszulesen und bereitzustellen. Weiterhin sollten die gewonnenen Informationen in einer Wrapperdatenstruktur gekapselt werden, um so ein weiteres einfacheres Verarbeiten der Informationen in den spezifischen Aktualisierungs- bzw. Generierungsklassen zu ermöglichen.
geplante Zeit: 15 Stunden
reale Zeit: Die geschätzte Zeit konnte im Wesentlichen eingehalten werden, da bereits die benötigent Fähigkeiten im Release 3 gesammelt werden konnten. Die Hauptaufgabe bestand also darin, den Fokus auf die neuen Javasyntaxelemente zu lenken.

Lesen der Daten aus der vom Controller aufbereiteten Datenstruktur, und schreiben in den AST
Beschreibung: Hier war es das Ziel, die Manipulationen, die durch die Manipulation der 3D-Szene entstehen, mit den Daten des JDT zu synchronisieren. Hierfür wurde die im `CodeController` erzeugte Hilfsstruktur mit den Daten des ***EFFECTS***-Datenmodells analysiert und die aktualisierten Daten in das JDT-Datenmodell übertragen.
geplante Zeit: 10 Stunden. Die Eclipsehilfe stellt bereits Codefragmente für diese Aufgabe zur Verfügung.

reale Zeit: Die reell benötigte Zeit betrug fast doppelt so viel wie zunächst geschätzt. Der Grund dafür lag darin, dass sich die Informationen in der Eclipsehilfe im Nachhinein als nicht vollständig herausstellten. Außerdem wurde bei der zeitlichen Abschätzung nicht berücksichtigt, dass Änderungen sich nicht nur auf den Quellcode einer Klasse beziehen können, sondern sich durchaus auch auf das darunterliegende Dateisystem auswirken müssen. Wenn beispielsweise eine Klasse in ein anderes Paket geschoben würde, müsste nicht nur das `package`-Statement in der Java-Datei geändert werden, sondern die Datei auch in das entsprechende Verzeichnis verschoben werden, das das Paket repräsentiert. Hierbei wurde sich aufgrund von Zeitmangel nur auf die Änderungen einer Java-Datei (Codestruktur und Dateiname) beschränkt.

Lesen der Daten, die von dem Controller aus dem AST bereitgestellt werden, und speichern in das *EFFECTS*-Datenmodell

Beschreibung: Dieser Task sollte es ermöglichen, die aktuellen Daten des JDT, die der `CodeController` bereitstellt, aufzubereiten und damit das ***EFFECTS***-Datenmodell zu füllen. So sollte es dann möglich sein, ein beliebiges Java-Projekt einzulesen und als dreidimensionale Szene darzustellen.

geplante Zeit: 10 Stunden. Für das Testen der Funktionalität und einer erwarteten hohen Anzahl von Problemfällen wurde relativ viel Zeit veranschlagt.

reale Zeit: 14 Stunden. Die geschätzte Zeit wurde um fast die Hälfte überschritten, da das Testen nicht so realisiert werden konnte wie geplant. Dazu kam es aufgrund von Kommunikationsschwierigkeiten zwischen den Entwicklern des Datenmodells und der Codegenerierung, bspw. wurden zeitweise die falschen Aktualisierungsmethoden für das Datenmodell verwendet.

Inspekor (Propertybox)

***Property View* anlegen**

Beschreibung: Der bereits in *Eclipse* existierende *Property View* sollte mit den diagrammspezifischen Daten erweitert werden.

geplante Zeit: 10 Stunden

reale Zeit: 14 Stunden. Der Mechanismus zur Benutzung des *Property Views* war schlecht dokumentiert.

Namen für Pakete, Klassen und Interfaces im *Property View*

Beschreibung: Der *Property View* soll die Namen für Pakete, Klassen und Interfaces anzeigen und editierbar machen können.

geplante Zeit: 5 Stunden
reale Zeit: 2 Stunden. Die Funktionalität konnte aus den vorherigen Releases übernommen werden.

Assoziationskardinalitäten im *Property View*
Beschreibung: Mit Hilfe des *Property Views* sollten Kardinalitäten an den Assoziationen gesetzt werden.
geplante Zeit: 5 Stunden
reale Zeit: 3 Stunden

Eintragen der Eigenschaften aus dem Inspektor ins Datenmodell
Beschreibung: Änderungen im *Property View* sollen ins Datenmodell übernommen werden, so dass sie in der Szene angezeigt werden.
geplante Zeit: 5 Stunden
reale Zeit: 5 Stunden

Proxies darstellen

Berechnung der Proxies
Beschreibung: Bei Erbung über Paketgrenzen hinweg sollten sogenannte Proxies benutzt werden, um die Klassen, die nicht in dem jeweiligen Paket sind, dort anzeigen zu können.
geplante Zeit: 20 Stunden
reale Zeit: Dieser Task wurde aus Zeitmangel nicht bearbeitet.

Graphische Repräsentation der Proxies erstellen
Beschreibung: Proxies sollten sich in der Darstellung von den in einem Paket tatsächlich vorhanden Klassen unterscheiden.
geplante Zeit: 2 Stunden
reale Zeit: Da die Proxies nicht implementiert wurden, wurde auch dieser Task nicht bearbeitet.

Graphische Feinheiten und Benutzbarkeit

Ausblenden der Vorderseiten von Paketwürfeln
Beschreibung: Je nach Kameraposition soll die Vorderseite eines Paketes ausgeblendet werden.

geplante Zeit:	1 Stunde
reale Zeit:	1 Stunde

Teilinformationen ein- und ausblenden

Beschreibung:	Es sollte die Möglichkeit gegeben sein Teilinformationen, wie z.B. Kardinalitäten oder Klassennamen, ein- und auszublenden.
geplante Zeit:	8 Stunden
reale Zeit:	Aufgrund der niedrigen Priorität und der mangelnden Zeit wurde dieser Task ausgelassen.

Erweiterung der Eclipseeinstellungen

Beschreibung:	Es sollte eine Seite bei den Eclipseeinstellungen geben, in der man die Darstellungseigenschaften einstellen kann.
geplante Zeit:	10 Stunden
reale Zeit:	Nach fünf Stunden wurde dieser Task aus den selben Gründen wie der vorherige Task abgebrochen.

Lotwurfverfahren

Beschreibung:	Für ein ausgewähltes Objekt sollte entweder ein Schatten auf den Boden oder Lote in alle drei Richtungen projiziert werden.
geplante Zeit:	3 Stunden
reale Zeit:	Im Verschiebemodus hat das zu verschiebene Objekt ein Lot zum Boden, die Schatten wurden weggelassen. In der Hälfte der Zeit wurde dieser Task erledigt.

Verändern des Mauszeigers

Beschreibung:	Der Mauszeiger sollte sich beim Wechsel in einen anderen Modus, wie z.B. in den Verschiebemodus, verändern.
geplante Zeit:	2 Stunden
reale Zeit:	Aufgrund der niedrigen Priorität und aus Zeitmangel wurde dieser Task ausgelassen.

Menüleiste in *Eclipse* einfügen

Beschreibung:	Die in der ***EFFECTS***-Perspektive sichtbare Menüleiste sollte in die Werkzeugleiste von *Eclipse* eingebunden werden.
geplante Zeit:	6 Stunden
reale Zeit:	4 Stunden

Assoziationen

Verschiedene Arten von Assoziationen anlegen und verwalten

Beschreibung:	Neben gerichteten und ungerichteten Assoziationen sollten auch noch Erbungs- und Implementierungsbeziehungen vorhanden sein.
geplante Zeit:	10 Stunden
reale Zeit:	8 Stunden

Übergeordnete Assoziationen zwischen Paketen

Beschreibung:	Übergeordnete Assoziationen zwischen Paketen sollten durch Sammelpipes angezeigt werden.
geplante Zeit:	15 Stunden
reale Zeit:	Dieser Task wurde aufgrund der niedrigen Priorität und aus Zeitmangel weggelassen.

17.4.3 Fazit

Im Wesentlichen wurden für das Release 4*a* alle Tasks abgearbeitet, bis auf die Tasks „Ausblendung der Vorderseiten von Information Cubes“, „Virtuelle Klassen (Proxies) sollen nicht auswählbar sein“ und „Kardinalitäten“. Diese drei Tasks wurden in das Release 4*b* übernommen und bearbeitet. Die geschätzte Gesamtzeit für diesen Task lag bei 161 Stunden. Letztendlich wurden aber 226 Stunden für Release 4*a* benötigt. Dies lag vor allem an den Tasks „Assoziationen“, „Kontrollerkonzept erstellen“ und „Objekte verschieben“, die sich als technisch schwierig herausstellten. Für das Release 4*b* kann man zusammenfassend sagen, dass fast alle Tasks erfolgreich abgearbeitet wurden. Insgesamt wurden für dieses Release ca. 255 Stunden benötigt, geplant waren dagegen ca. 336 Stunden. Aufgrund der mangelnden Zeit konnten die folgenden Tasks nicht abgearbeitet werden, die alle zusammen ca. 75 Stunden Zeitersparnis einbrachten. Der Task „Syntaxcheck auf graphischer Ebene realisieren“ und der damit zusammenhängende Task „Kollisionsabfrage“ wurden wegen zeitlicher und technischer Probleme weggelassen. Des Weiteren wurden diese Tasks mit niedriger Priorität ebenfalls nicht bearbeitet: „Berechnung der Proxies“, „Graphische Repräsentation der Proxies erstellen“, „Verändern des Mauszeigers“ und „Übergeordnete Assoziationen zwischen Paketen“.

17.5 Vorstellung der implementierten Architektur

Stephan Eisermann, Jan Wessling, Sven Wenzel

Im Nachfolgenden soll die in diesem Release implementierte Architektur vorgestellt werden. Dazu wird zunächst die geplante Architektur beschrieben, im Anschluss daran die realisierte Architektur geschildert. Abschließend werden beide verglichen.

17.5.1 Beschreibung der geplanten Architektur

Die von uns geplante Architektur gliedert sich in mehrere Teile.

Das Datenmodell

Als erstes ist das für das Plugin spezifische Datenmodell zu nennen. Dieses gliedert sich in die zwei Teile JDT-Erweiterung mit Containern und in die Klassen die nicht direkt an das JDT gebunden sind.

Wie in den vorangegangenen Releases auch wurde das von dem Plugin benötigte Datenmodell als Erweiterung der JDT-Klassen unter der Zuhilfenahme von Containern geplant. Hierbei handelt es sich um die Klassen `CPDInterfaceConatiner`, `CPDClassContainer`, `CPDPackageContainer` und `CPDProjectContainer`.

Es zeigte sich, dass die für das Plugin nötige Funktionalität nicht alleine durch die Container zu realiseren war. Aus diesem Grund waren noch weitere Klassen für Assoziationen und Subpaketstrukturen, wie `AssociationElement`, `ImplementsElement`, `ExtendsElement` und `CPDSubPackageContainer` nötig, die nicht direkt mit dem Eclipse-Datenmodell verbunden sind, da es für sie keine Äquivalente im JDT gibt. Die Struktur der Subpakete ist durch den `CPDSubPackageContainer` gegeben. Hierzu hat der `CPDSubPackageContainer` eine Referenz auf das Oberpaket, sowie auf alle seine Kindelemente. Relationen werden direkt mit dem `CPDProjectContainer` assoziert, da sie nicht immer eindeutig einem bestimmten Paket zuzuordnen sind. Die geplante Architektur der Datenmodells zeigt Abb. 17.1.

Der Zustandsautomat

Als zweites ist das Modell des Zustandsautomaten zu nennen. Das vierte Release unterscheidet sich insofern von den vorangegangenen Releases, als dass hier Interaktion vorgesehen ist und somit auf Nutzereingaben reagiert werden muss.

Das Verhalten der GUI wird durch einen Zustandsautomaten definiert. Als Grundlage für den Zustandsautomaten wurden aus dem Artikel (Yacoub und Ammar, 1998) die Patterns „Meally“, „Interface“, „State Driven“ und „Exposed“ genommen. Es wurde geplant, dass der Hauptteil des Automaten im Core-Plugin realisiert wird. Die Klasse `FSMInterface` ist hier der Kern des Automaten.

Sie implementiert diverse Listener (bspw. Mouse- und Keyboardlistener) und bietet Methoden zum expliziten Methodenaufrufen durch Benutzereingaben. Somit ist eine Schnittstelle zwischen dem eigentlichen Plugin und dem Zustandautomaten realisiert, welches Reaktionen sowie explizite Aufrufe entgegennimmt und an den aktuellen Zustand weiterreicht. Für das `FSMInterface` ist es transparent, welcher Zustand der Aktuelle ist. Jeder spezielle Zustand muss von `FSMState` erben, und realisert die unterschiedlichen Reaktionen auf Nutzereingaben.

Als Standardmodus ist der `CameraMode` vorgesehen. Dieser beschreibt ein Standardverhalten und sollte von den spezifischen Diagrammtypen überschrieben werden, daher ist als Eclipse-Erweiterungspunkt realisiert. Als spezifische Zustände für das Plugin wurden nach der Modellierung des Zustandsautomaten die Klassen `RemoveMode`, `InsertMode`, `ObjectSelectedMode`, `MoveMode`, `AssociationMode` identifiziert. Abbildung 17.2 zeigt das Klassendiagramm des Zustandsautomaten.

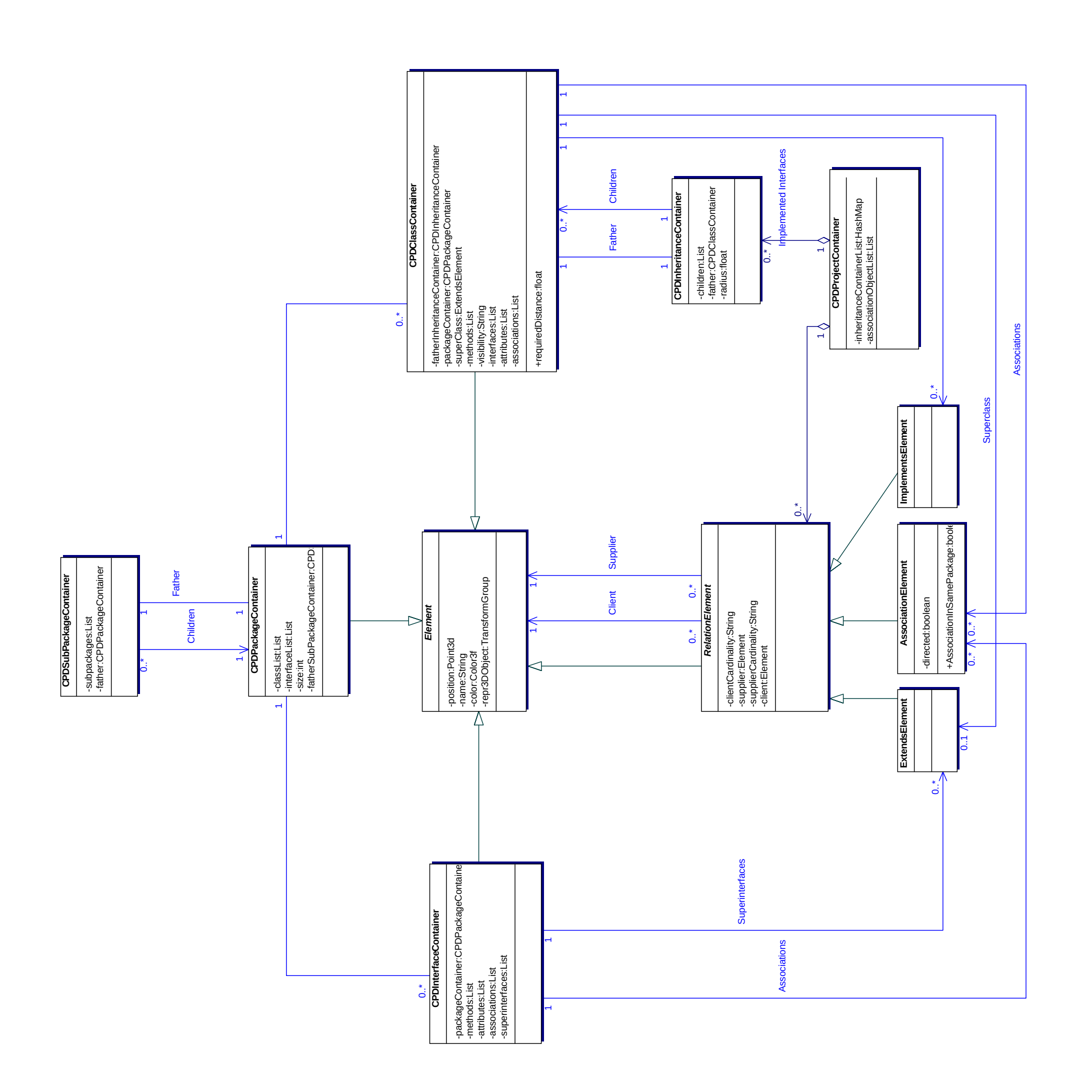


Abbildung 17.1.: Datenmodell

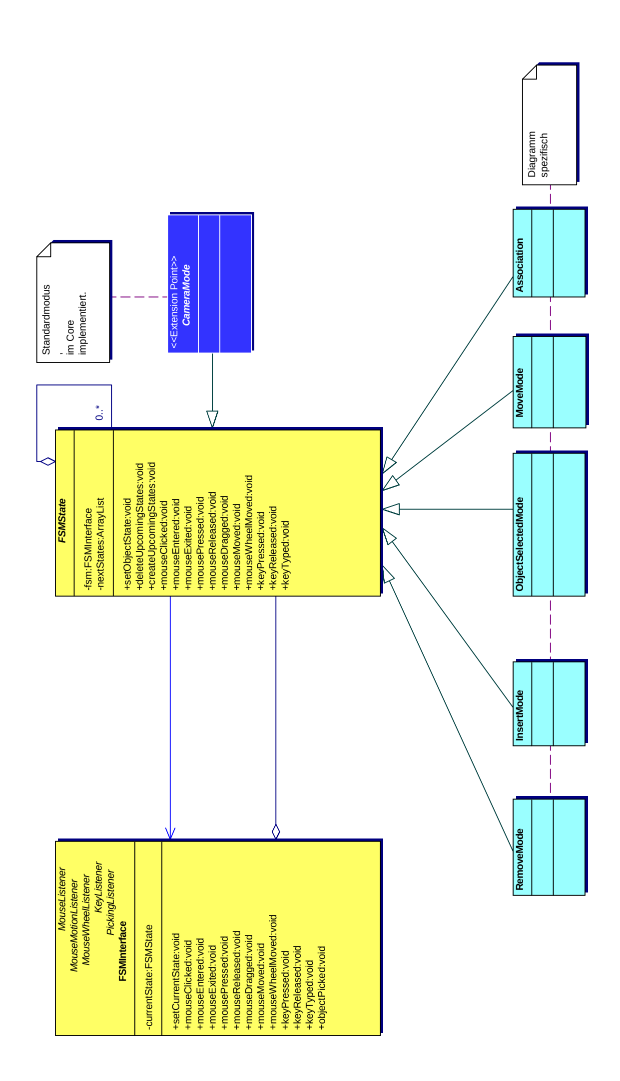


Abbildung 17.2.: Geplanter Zustandsautomat

Ein UML–Zustandsdiagramm (siehe Abb. 17.3) beschreibt das geplante Verhalten des Automaten. Der `CameraMode` ist der Ausgangspunkt für alle Aktionen. Von hier aus können alle anderen Modi erreicht werden. Hier kann man grob zwischen Einfügemodi und dem Selektionsmodus unterscheiden. Aus dem Selektionsmodus können Objekte gelöscht oder verschoben werden. Genaue Details des Zustandsautomaten können der Abbildung 17.3 entnommen werden.

Die Plugin Kontrollstruktur

Zuletzt musste der interne Aufbau des Projekts angepasst werden. Der bisherige Aufbau setzte vorraus, dass nur zum Startzeitpunkt ein Diagramm berechnet wurde. Entsprechend war bisher die Struktur realisiert. Im vierten Release ist dieses Vorgehen so nicht mehr durchführbar, da durch Benutzereingabe eine wiederholte Berechnung der Anzeige nötig wurde. Der View-Controller, welcher bisher die Fragment-interne Koordination vorgenommen hat, sollte durch einen Editor ersetzt werden. Gleichzeitig wurde so die Eclipseintegration verbessert, weil jede Modifikation in Eclipse über einen Editor realisiert wird. Im Zuge des Refactorings wurde eine strikte Trennung von Model und View angestrebt. Diese war in bisherigen Releases nicht gegeben.

17.5.2 Beschreibung der realisierten Architektur

Das Datenmodell

Das realisierte Datenmodell orientiert sich stark an der von Eclipse bereitgestellten Struktur. So gibt es für Klassen, Pakete, Interfaces und für das Projekt im JDT jeweils ein Äquivalent in unserem Datenmodell. Gleichzeitig gibt es noch Klassen, die Relationen darstellen, allerdings keinen Klassen im JDT zugeordnet werden können. So gibt die Klasse `CPDInheritanceContainer` die Vererbungshierarchie von einer Klasse und ihren Kindern wieder. `AssociationElement`, `ExtendsElement` und `ImplementsElement`, die alle von `RelationElement` erben, geben die einzelnen Relationen zwischen zwei Elementen wieder. Alle Relationen werden im `CPDProjectContainer` gehalten.

Jedes Paket verwaltet eine Liste seiner Unterpakete. Hierdurch ist implizit die Struktur zwischen Paketen und ihren Unterpaketen gegeben. Eine Übersicht über das Datenmodell ist in Abb. 17.4 zu sehen.

Das Datenmodell wird durch eine Fassade gekapselt, um einen einheitlichen Zugriff zu gewährleisten und das MVC-Konzept zu realisieren.

Der Zustandsautomat

Die Struktur des Zustandsautomaten wurde wie geplant und in 17.5.1 beschrieben umgesetzt. Das zugehörige Zustandsdiagramm wurde im Zuge der Umsetzung den Erfordernissen entsprechend verfeinert. Der `CameraMode` ist weiterhin der Einstiegspunkt des Automaten. Von hier aus kann man die vier Einfügemodi `PackageInsertMode`, `ClassInsertMode`, `InterfaceInsertMode` und `AssociationInsertMode` erreichen. Außerdem ist der Übergang in den `ObjectSelectedMode` möglich. Aus diesem Zustand kann man weitere Zustände erreichen, um Objekte zu löschen oder zu verschieben. Ist ein Paket selektiert, so ist es möglich, vom `ObjectSelectedMode` aus die `Interface-`, `Class-`, und `PackageInsertModi`

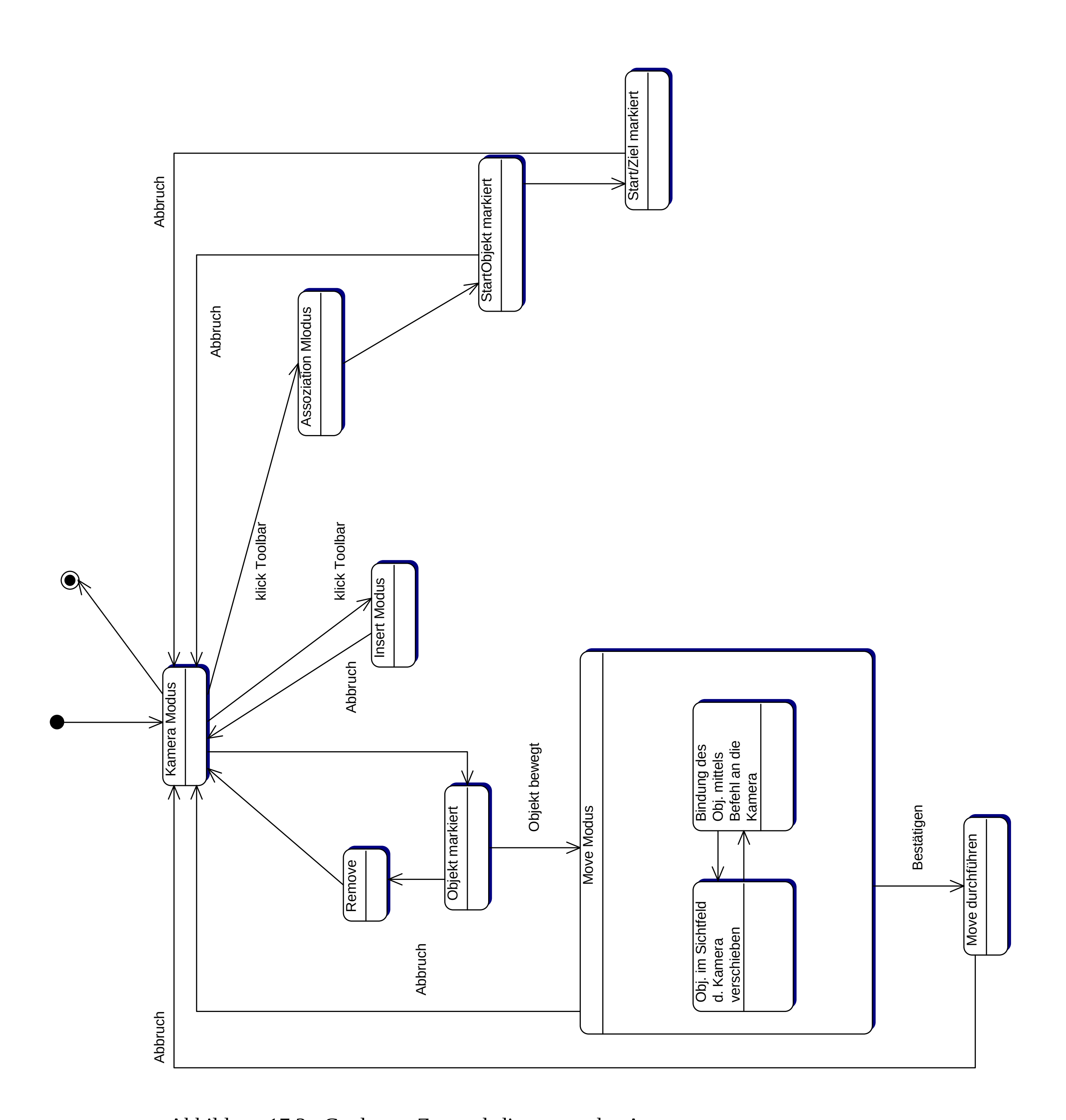


Abbildung 17.3.: Geplantes Zustandsdiagramm des Automaten

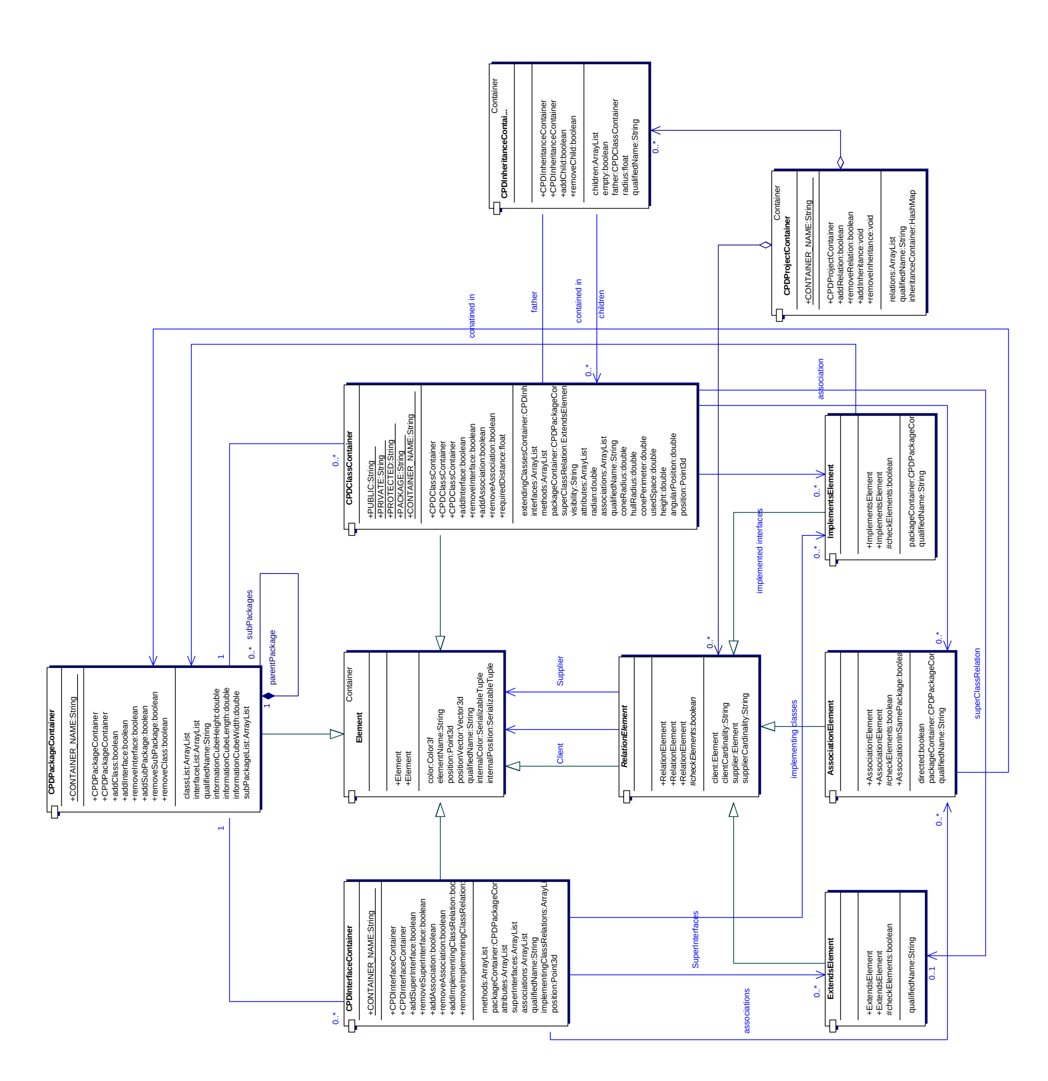


Abbildung 17.4.: Realisiertes Datenmodell

zu erreichen. Dies bedeutet, dass man neue Klassen, Interfaces und Pakete auch in bestehenden Paketen erzeugen kann.

Beim Verschieben kann man wahlweise ein Objekt in dem momentan sichtbaren Raum verschieben, oder es im `ObjectMovementwithCamera`-Mode an die Kamera binden und das Objekt so über größere Strecken im Raum bewegen.

Abb. 17.5 zeigt den realisierten Zustandsautomaten.

Die Plugin Kontrollstruktur

Es gibt nun zwei Möglichkeiten, das Plugin anzustoßen und ein Diagramm zu erstellen. Auf der einen Seite gibt es den `NewDiagramWizard`, der beim Erstellen eines neuen und leeren Diagramms aufgerufen wird, und auf der anderen Seite den Menüpunkt *Generate ClassPackageDiagram from Project*. Von diesen beiden wird die Generierung des `ClassPackageDiagramms` angestoßen. Dabei wird der `CPDInitializer`, welcher von `PluginInitializer` erbt, erzeugt. Dieser Initializer hat die Aufgabe, die grundlegende Initialisierung des Fragments vorzunehmen, also einen Controller zu instanziieren und als Verwalter für die aktuelle Diagrammdatei zu registrieren. Dieser `CPDController`, eine Spezialisierung des `PluginControllers`, stellt den eigentlichen Controller im Sinne des MVC-Konzepts dar. Er kennt auf der einen Seite die `CPDScene`, also den View, und auf der anderen Seite die `DataModelFacade`, welche den Datenanteil des MVC-Konzeptes realisiert und den Zugriff, wie in 17.5.2 beschrieben, kapselt. Änderungen im Datenmodell werden durch den Aufruf der Methode `updateView` im `Editor` durch den `CPDController` in den View propagiert. Ausserdem hat er eine Verbindung zum `FSMInterface`. Das `FSMInterface` wurde in Kapitel 17.5.2 näher beschrieben. Es besitzt verschiedene `Listener`, die die Scene überwachen, und ist somit der Part, der den Kontrollfluß zwischen `Scene` und `CPDController` regelt.

Die realisierte Pluginstruktur in in Abb. 17.6 dargestellt.

17.5.3 Vergleich zwischen geplanter und realisierter Architektur

Das Datenmodell

Die realisierte Architektur unterscheidet sich nur in wenigen Fällen von der geplanten Architektur. So gibt es nur zwei nennenswerte Unterschiede. Zum einen ist der `CPDSubPackageContainer` durch eine Liste im `CPDPackageContainer` ersetzt worden, die Referenzen auf die Unterpakete enthält. Zum anderen ist das Datenmodell durch den Controller gekapselt worden. Die zweite Änderung geschah aufgrund der Realisierung des MVC-Konzepts, damit sämtliche Zugriffe auf das Datenmodell durch den Controller gesteuert werden können.

Der Zustandsautomat

Der Zustandsautomat wurde in der realsierten Version bezüglich der Namensgebung der Zustände überarbeitet. Die verschiedenen Einfügemodi sind nun einheitlich benannt. Die generelle Trennung in Einfügemodi und einen Selektionsmodus wurde beibehalten. Es zeigte sich, dass die Möglichkeit, einem Paket Unterpakete und Klassen hinzufügen zu können, benötigt wird. Entsprechend wurde der `ObjectSelectedMode` so erweitert, dass von hier aus die ent-

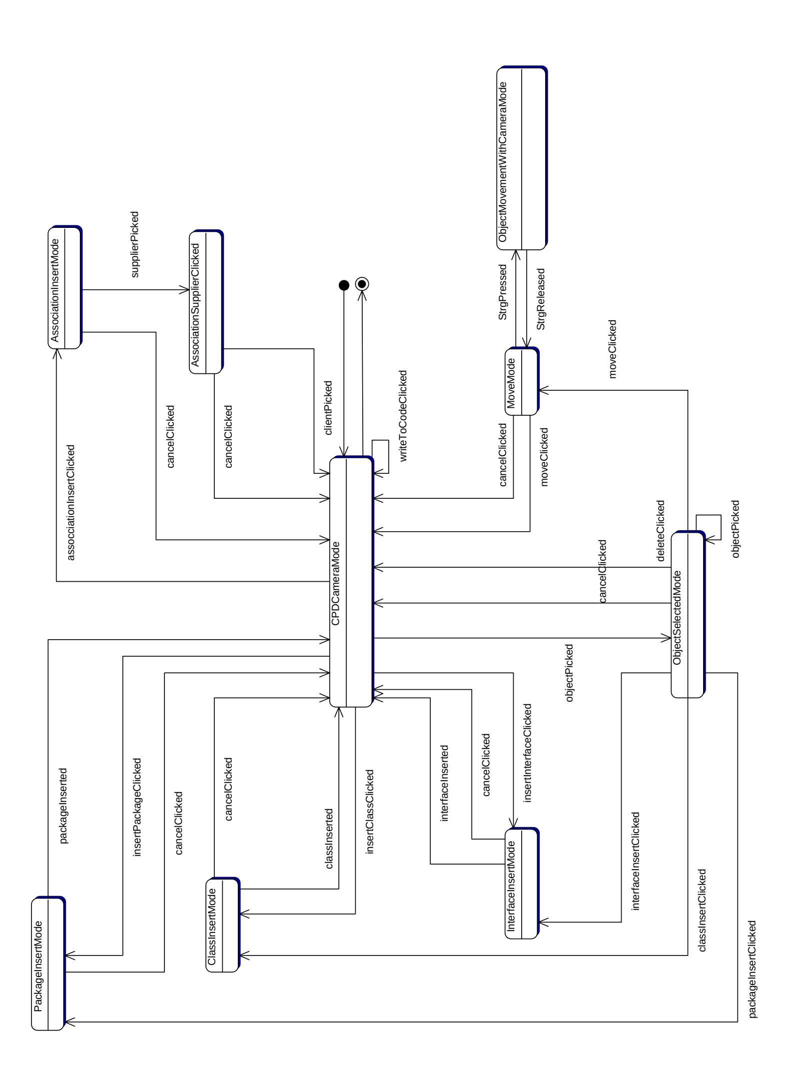


Abbildung 17.5.: Realisiertes Zustandsdiagramm des Automaten

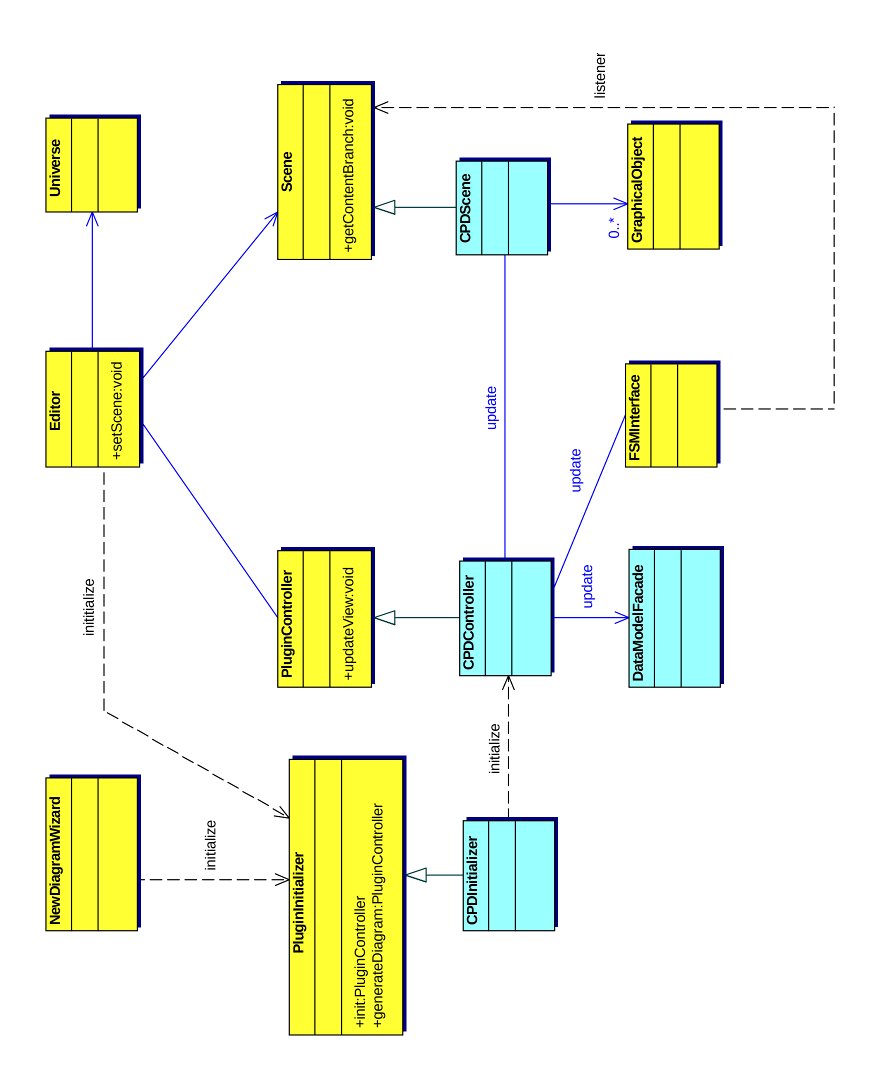


Abbildung 17.6.: Die realisierte Pluginstruktur

sprechenden Einfügemodi erreichbar sind.

Die Plugin Kontrollstruktur

Die Planung ging nur von einer strikteren Trennung von Model und View aus. Die Struktur, wie sie sich jetzt mit einem echten Controller als Schicht zwischen Model und View darstellt, ergab sich erst während der Realisierungsphase. Die strikte Trennung von Model und View konnte durch diesen Controller, der die Zugriffe auf das Datenmodell kapselt, umgesetzt werden. Datenmodell und graphische Objekte sind nun strikt getrennt und nur über den `Controller` verbunden. Ebenso wurde die Eclipseintegration durch die Verwendung eines Editor verbessert.

17.5.4 Fazit

Die Modellierung von Datenmodell und Zustandsautomat in der Planungsphase erwies sich als ausreichend. Fast alle benötigten Eigenschaften konnten schon in der Planungsphase defininert werden, so dass die Umsetzung ohne große Änderungen vonstatten gehen konnte.

Bei der Umsetzung des MVC-Konzepts war eine Planung aufgrund der Unkenntnis der genauen Realisierung von *Eclipse*-Editoren und einem größerem Refactoringaufwand nicht möglich. So geschah die Planung iterativ und passte sich mehrmal den technischen Gegebenheiten von Eclipse an.

17.6 Kundenakzeptanztest

Armin Bruckhoff, Sven Wenzel, André Kupetz, Michael Nöthe

Da zwei offizielle Releaseabnahmen stattfanden, werden an dieser Stelle die Akzeptanztests und Ergebnisse für beide Abnahmen aufgeführt.

17.6.1 Akzeptanztest Release 4a

In diesem Abschnitt werden die Abnahmetests beschrieben, so wie sie aus den User Stories hervorgehen. In diesem Release war es allerdings aufgrund der vorwiegend visuellen Anforderungen nicht möglich, die Testfälle zu automatisieren. Folglich wurden nur visuelle Tests durchgeführt.

Ein Objekt soll sichtbar markiert werden können.
Testergebnis: bestanden
Anmerkung: Interfaces sind nicht auswählbar.

Mehrere Objekte sollen sichtbar markiert werden können.
Testergebnis: bestanden
Anmerkung: Interfaces sind nicht auswählbar.

Die ausgewählten Objekte sollen verschoben werden können.
Testergebnis: bestanden
Anmerkung: Assoziationen zwischen Klassen werden nicht mit verschoben. Das Verschieben von Klassen, die in einem Paket liegen, das selbst nicht im Ursprung liegt, liefert falsche Ergebnisse. Die Kamera muss nach dem Beginn des Verschiebevorgangs einmal kurz bewegt werden, um eine normale Verschiebegeschwindigkeit zu erhalten.

Die ausgewählten Objekte sollen gelöscht werden können.
Testergebnis: bestanden
Anmerkung: –

Die Eigenschaften ausgewählter Objekte sollen verändert werden können.
Testergebnis: nicht bestanden
Anmerkung: Diese Anforderung wurde aufgrund des benötigten Inspektors ins nächste Release verschoben.

Klassen sollen hinzugefügt werden können und durch Würfel dargestellt werden.
Testergebnis: bestanden
Anmerkung: Die erstellten Klassen werden immer im Ursprung angelegt.

Interfaces sollen hinzugefügt werden können und durch Kugeln dargestellt werden.
Testergebnis: bestanden
Anmerkung: Die erstellten Interfaces werden immer im Ursprung angelegt.

Pakete sollen hinzugefügt werden können und durch halbtransparente Würfel dargestellt werden.
Testergebnis: bestanden
Anmerkung: Die erstellten Pakete werden immer im Ursprung angelegt.

Erbungshierarchien sollen durch Cone Trees abgebildet werden.
Testergebnis: nicht bestanden
Anmerkung: Cone Trees sind in diesem Release noch nicht implementiert werden.

Wenn eine Klasse auf einen Cone Tree gezogen wird, soll der Cone Tree neu angeordnet werden.

Testergebnis: nicht bestanden
Anmerkung: Aufgrund der fehlenden Cone Trees (s.o.) konnte dieses Verhalten nicht erfüllt werden.

Wenn eine Erbungsassoziation eingefügt wird, soll der Cone Tree neu angeordnet werden.

Testergebnis: nicht bestanden
Anmerkung: Auch dieser Test konnte aufgrund der fehlenden Cone Trees noch nicht abgenommen werden.

Für Vererbungshierarchien über Paketgrenzen hinweg sollen Proxies die Darstellung im Fremdpaket übernehmen.

Testergebnis: nicht bestanden
Anmerkung: Diese Anforderung wurde aufgrund eines Ressourcenengpasses ins nächste Release verschoben.

Auf dem Boden soll für jedes Objekt ein Schatten dargestellt werden, um so die Höhenposition besser abschätzen zu können.

Testergebnis: nicht bestanden
Anmerkung: Diese Anforderung wurde aufgrund eines Ressourcenengpasses ins nächste Release verschoben.

Eine Werkzeugleiste ermöglicht die Auswahl der Bearbeitungsmodi.

Testergebnis: bestanden
Anmerkung: Die Werkzeugleiste sollte, sobald der Editor immer integriert angezeigt werden kann, in eine *SWT-Toolbar* umgewandelt werden.

Bearbeitungsmodi sollen in einer Menüleiste ausgewählt werden können.

Testergebnis: nicht bestanden
Anmerkung: Diese Anforderung ist nicht umgesetzt worden, da sich eine Werkzeugleiste als sinnvoller erwiesen hat.

Zwischen den dargestellten Objekten sollen Assoziationen erstellt werden können.

Testergebnis: bestanden

Anmerkung: Es ist nicht möglich, Assoziationen von und zu Interfaces zu erstellen.

Zwischen den dargestellten Paketen sollen größere Assoziationen die Assoziationen, die zwischen den Klassen und Interfaces dieser Pakete existieren, ersetzen.
Testergebnis: nicht bestanden
Anmerkung: Aufgrund mangelnder Zeit wurde dieser Task nicht umgesetzt.

Anhand der Farbe soll der Typ einer Assoziation erkennbar sein.
Testergebnis: nicht bestanden
Anmerkung: Es gab in diesem Release nur einen Assoziationstyp, deshalb ist es nicht möglich gewesen, verschiedene Assoziationstypen auszuwählen bzw. darzustellen.

Assoziationen sollen mit Kardinalitäten beschriftet sein.
Testergebnis: nicht bestanden
Anmerkung: Es war in diesem Release noch keine Möglichkeit gegeben, Kardinalitäten einzugeben.

17.6.2 Akzeptanztest Release 4b

In diesem Abschnitt werden die Abnahmetests beschrieben, so wie sie aus den User Stories des Releases 4*b* hervorgehen. Auch in diesem Release war es nicht möglich, die Testfälle zu automatisieren. Die wenigen Kriterien, die man hätte formalisieren können (wie z.B. das Enthaltensein eines Klassenwürfels in einem Paket) konnten geeigneter durch visuelle Test überprüft werden, ebenso wie die ohnehin visuellen Anforderungen. Daher wurden nur visuelle Tests durchgeführt. Die Tests gliedern sich in zwei Abschnitte. Zunächst wird ein leeres Projekt angelegt und mittels des graphischen Editors gefüllt. Im zweiten Abschnitt wird ein bestehendes Projekt mit Hilfe des ***EFFECTS***-Plugins eingelesen und graphisch dargestellt.

Die Toolbar zur graphischen Manipulation soll in die *Eclipse-Toolbar* eingebettet sein.
Testergebnis: bestanden
Anmerkung: –

Ein neues Projekt soll angelegt werden können.
Testergebnis: bestanden
Anmerkung: –

Es soll ein Paket eingefügt werden. Bei der Darstellung dieses Paketes sollen die dem Betrachter zugewandten Wände ausgeblendet sein.

Testergebnis: bestanden

Anmerkung: Nach diesem Test sollte auf Wunsch der Geschäftsleitung eine Klasse mit Hilfe der Toolbar in das Diagramm eingefügt werden. Ein gerade erstelltes Paket ist automatisch selektiert, wodurch eine neue Klasse dem Paket direkt hinzugefügt wurde. Die Klasse wurde jedoch an der akuellen Kameraposition eingefügt, was dazu führte, dass der Paketwürfel drastisch vergrößert wurde, um die Klasse mit einzuschließen. Hier wurde angemerkt, dass stattdessen besser die Klasse in den Paketwürfel verschoben werden sollte, damit dieser nicht zu groß wird.

Der Mauszeiger soll sich beim Wechsel von Bearbeitungsmodi verändern.

Testergebnis: nicht bestanden

Anmerkung: Aufgrund der niedrigen Priorität und des Zeitmangels wurde dieser Task nicht umgesetzt.

Es soll eine Klasse in das Diagramm eingefügt werden. Diese Klasse soll automatisch benannt und der Name im Diagramm angezeigt werden.

Testergebnis: bestanden

Anmerkung: –

Es soll nicht möglich sein, Elemente wie Klassen und Interfaces überlappend in der Szene anzuordnen.

Testergebnis: nicht bestanden

Anmerkung: Für diese Funktion fehlte die Kollisionsabfrage.

Die im vorherigen Test eingefügte Klasse soll in das ebenfalls vorher angelegte Paket verschoben werden können. Nach der Betätigung des „Write To Code“-Buttons soll die Paketdeklaration im Sourcecode der Klasse erscheinen.

Testergebnis: nicht bestanden

Anmerkung: Für diese Funktion fehlte die Kollisionsabfrage.

Wenn beim Verschieben einer Klasse der Klassenwürfel teilweise innerhalb und teilweise außerhalb eines Paketwürfels abgelegt wird, soll die Syntaxprüfung dieses als Fehler anzeigen bzw. die Verschiebung zurückgesetzt werden.

Testergebnis: nicht bestanden

Anmerkung: Für diese Funktion fehlte die Kollisionsabfrage.

Für ein ausgewähltes Objekt soll entweder ein Schatten auf den Boden oder Lote in alle drei Richtungen projiziert werden.
Testergebnis: bestanden
Anmerkung: Es wurden Lote projiziert.

Beim manuellen Einfügen einer Erbungsbeziehung soll der Cone Tree automatisch angeordnet werden.
Testergebnis: bestanden
Anmerkung: –

Beim Verschieben einer Klasse, die sich in einem Cone Tree befindet, soll die Klasse vom Cone Tree abgekoppelt werden und mit ihrem Unterbaum an die neue Position verschoben werden.
Testergebnis: bestanden
Anmerkung: –

Nach dem graphischen Aufbau eines Cone Trees und dem Betätigen des „Write To Code“-Buttons soll sich die Erbungsbeziehung im Code der beteiligten Klassen niederschlagen.
Testergebnis: bestanden
Anmerkung: Ebenso funktioniert jetzt das Umbenennen einer Klasse im *Property View* und das Schreiben des neuen Namens in den Code.

Der Versuch, eine zyklische Erbung oder eine Mehrfacherbung in das Diagramm einzufügen, soll fehlschlagen.
Testergebnis: bestanden
Anmerkung: Sowohl die zyklische Vererbung als auch die Mehrfacherbung werden mit der gleichen Fehlermeldung abgelehnt. Wenn möglich, sollen die beiden Fehler jedoch unterschieden werden können.

Es soll eine Seite bei den Eclipseeinstellungen geben, mit deren Hilfe die Darstellung verschiedener Details ein- und ausgeschaltet werden kann.
Testergebnis: nicht bestanden
Anmerkung: Aufgrund der niedrigen Priorität und des Zeitmangels wurde dieser Task nicht umgesetzt.

Es soll möglich sein, mit Hilfe eines *Property Views* Eigenschaften von gewählten Elementen, wie z.B. Namen, Kardinaltäten, Paketzugehörigkeiten, Erbungsbeziehungen, zu editieren.

Testergebnis: bestanden
Anmerkung: Es fehlten die Einstellungsmöglichkeiten für Erbungsbeziehungen und Paketzugehörigkeit im *Property View*. Namen und Farben konnten jedoch bearbeitet werden. Es wurde angemerkt, dass Kardinalitäten und Kantenbezeichnungen für Erbungskanten keinen Sinn machen und die entsprechenden Felder im *Property View* für solche Kanten ausgeblendet werden sollten.

Es soll verschiedene, farblich unterscheidbare Pfeile für ungerichtete und gerichtete Assoziationen, Erbung und Implementierung geben.

Testergebnis: bestanden
Anmerkung: –

Assoziationen sollen mit Kardinalitäten beschriftet sein.

Testergebnis: bestanden
Anmerkung: Im Diagramm sind die Positionen von Supplier und Client vertauscht.

Übergeordnete Assoziationen zwischen Paketen sollen angezeigt werden.

Testergebnis: nicht bestanden
Anmerkung: Aufgrund der niedrigen Priorität und des Zeitmangels wurde dieser Task nicht umgesetzt.

Es soll möglich sein, die erstellte Szene zu speichern, das Editorfenster zu schließen und danach die Szene wiederherzustellen.

Testergebnis: bestanden
Anmerkung: –

Die nun folgenden Tests werden auf einem zuvor erstellten Testprojekt durchgeführt. Es wird also vorhandener Sourcecode eingelesen und dargestellt.

Für das vorhandene Projekt soll eine *.efx-Datei angelegt und damit eine Szene generiert werden. Die enthaltenen Elemente sollen dabei sinnvoll angeordnet werden.

Testergebnis: bestanden

Anmerkung: Das Anlegen eines neuen Projektes gestaltet sich etwas umständlich, da nach dem Erstellen der *.efx-Datei noch das eigenliche Diagramm generiert werden muss. Außerdem funktioniert das automatische Platzieren von Cone Trees nicht.

Es wird eine neue Klasse mit Hilfe des *Eclipse-Wizards* angelegt, die von einer vorhandenen Klasse des Projektes erbt. Nach erneuter Generierung des Diagramms soll diese Klasse in dem entsprechenden Cone Tree angezeigt werden.
Testergebnis: bestanden
Anmerkung: –

Es wird ein neues Interface mit Hilfe des *Eclipse-Wizards* angelegt, das Teil eines vorhandenen Paketes des Projektes sein soll. Nach erneuter Generierung des Diagramms soll dieses Interface in dem entsprechenden Paketwürfel angezeigt werden.
Testergebnis: bestanden
Anmerkung: –

Es wird eine neue Klasse mit Hilfe des *Eclipse-Wizards* angelegt, die mit einem vorhandenen Interface des Projektes in einer Implementierungsbeziehung steht. Nach erneuter Generierung des Diagramms soll diese Klasse durch einen Implementierungspfeil mit dem entsprechenden Interface verbunden sein.
Testergebnis: bestanden
Anmerkung: Dies funktioniert sogar mit Assoziationen.

17.6.3 Anschließende Korrekturen

Im Anschluss an den Kundenakteptanztest zwischen den Releases 4*a* und 4*b*, sowie im Anschluss an Release 4*b* sind noch einige kleinere Fehler korrigiert worden. Nachfolgend sollen diese Korrekturen noch einmal kurz beschrieben werden.

- Die graphischen Figuren der Interfaces sind überarbeitet worden, so dass Interfaces nun auch selektierbar sind und sichtbar markiert werden können.

- Es werden nun verschiedene Assoziationstypen unterstützt. Dieses sind die ungerichtete und die gerichtete Assoziation sowie Implementations– und Erbungsassoziationen. Dabei sind die Farben für die verschiedenen Assoziationstypen vorgegeben, um eine visuelle Unterscheidung zu unterstützen.

- Die Visualisierung von Vererbungshierarchien werden nun durch Cone Trees realisiert.

Hierbei werden Klassen automatisch auf dem Cone Tree angeordnet, sobald entsprechende Erbungsassoziationen eingefügt worden sind.

- Der *Property Viewer*, der in Release 4*b* realisiert wurde, ermöglicht für gerichtete und ungerichtete Assoziationen die Angabe von Kardinalitäten, die im Diagramm entsprechend angezeigt werden.

TEIL 4

Beschreibung des Frameworks

KAPITEL 18

Einleitung

Sven Wenzel

18.1 Motivation

Eine dreidimensionale Darstellung von Softwarestrukturen ermöglicht die Veranschaulichung von Sachverhalten, die in herkömmlichen Diagrammen mit nur zwei Dimensionen nicht oder nur sehr schwierig zu zeigen sind. Des Weiteren können dreidimensionale Diagramme beispielsweise die Übersicht erhöhen und den Informationsgehalt steigern.

Als Beispiel kann man hierzu das UML-Sequenzdiagramm und das Diagramm aus Release 3 heranziehen. Es veranschaulicht, die Struktur der Methodenaufrufe zwischen verschiedenen Objekten. Bei einer großen Anzahl von Objekten wird dieses Diagramm sehr unübersichtlich, da die Anzahl der Aufrufpfeile zunimmt. Außerdem ist eine Anordnung, in der Objekte mit vielen Aufrufbeziehungen in räumlicher Nähe zueinander angeordnet werden, auf einer zweidimensionalen Fläche nur eingeschränkt möglich. Das *Sequencediagram* aus Release 3 nutzt den dreidimensionalen Raum aus, um Objekte, die durch viele Aufrufpfeile verbunden sind, räumlich nahe beieinander anzuordnen und Objekte mit weniger vielen Methodenaufrufen weiter entfernt zu positionieren. Darüberhinaus können die Aufrufpfeile kreuzungsfrei angeordnet werden, wodurch sich die Übersichtlichkeit erhöht.

Um dreidimensionale Diagramme sinnvoll für den im Softwareentwicklungsprozess einzusetzen, sollten diese in eine Entwicklungsumgebung integriert werden. So ist es möglich – integriert in den Entwicklungsprozess – Diagramme zu betrachten und zu bearbeiten, ohne das Werkzeug wechseln zu müssen.

18.2 Unterstützung durch das Framework

Zu der Entwicklung eines solchen Diagramms zählen insgesamt vier Kernaufgaben:

1. Entwurf des dreidimensionalen Diagramms, also verschiedene Symbole und eine Grammatik,

2. die technische Umsetzung der einzelnen Komponenten dieses Diagramms, also z.B. Figuren (Symbole) oder Anordnungsalgorithmen (Grammatikregeln),

3. die technische Umsetzung des Rahmens für das dreidimensionale Diagramm und der damit verbundenen Eigenschaften, wie z.B. der Navigation
   sowie

4. die Integration des Diagramms in die Entwicklungsplattform.

Die Entwicklung eines neuen dreidimensionalen Diagrammtyps wird durch die Verwendung des ***EFFECTS***-Framework entscheidend vereinfacht. ***EFFECTS*** stellt ein Applikationsrahmen bereit, der es erlaubt, Diagramme zur dreidimensionalen Visualisierung von Softwarestrukturen zu erzeugen und sie in die Entwicklungsplatform *Eclipse* zu integrieren.

Die Aufgaben des Entwicklers reduzieren sich hierbei auf die ersten beiden Kernaufgaben. Den Entwurf des Diagramms sowie die technische Realisierung der einzelnen Diagrammkomponenten kann das Framework dem Entwickler selbstverständlich nicht abnehmen. Die anderen beiden Aufgaben braucht der Entwickler aber nicht näher beachten. Er kann sich somit also auf seine eigentliche Aufgabe – den Entwurf eines neuen Diagrammtyps – beschränken.

Das ***EFFECTS***-Framework beinhaltet die Realisierung des gesamten Diagrammrahmens. Benötigte Funktionen, die mit der Dreidimensionalität einherkommen sind bereits umgesetzt. Ein Beispiel hierfür ist die Navigation durch den dreidimensionalen Raum – insbesondere unter Berücksichtigung, dass nur zweidimensionale Eingabegeräte zur Verfügung stehen.

Auch immer wiederkehrende Aufgaben, wie das Laden und Speichern von Diagramminstanzen, werden dem Entwickler abgenommen und sind bereits in das ***EFFECTS***-Framework integriert.

Darüber hinaus wurde ***EFFECTS*** als ein Plugin für die Entwicklungsplattform *Eclipse* implemetiert. Eng damit verbunden werden auch die Diagramme vollständig in die Benutzungsschnittstelle von *Eclipse* integriert. Die Arbeit mit den Diagrammen ist somit gänzlich in den Arbeitsprozess eines Anwenders integrierbar, ohne dass ein Wechsel des Werkzeuges erfolgen muss. Des Weiteren ermöglicht die Integration in *Eclipse* das Zusammenspiel zwischen den dreidimensionalen Diagrammen und zugrundeliegenden Informationen, die der Entwicklungsplattform vorliege und sinnvoll für das Diagramm zu verwenden sind.

Über den Diagrammrahmen und die IDE-Integration hinaus stellt ***EFFECTS*** bereits vollständig implementierte grafische Objekte zur Verfügung, die als Symbole in einem Diagramm verwendet werden können.

Der Entwickler eines neuen Diagramms kann also die meisten Umgebungseigenschaften vernachlässigen und das Augenmerk auf die Entwicklung des eigentlichen Diagramms legen, wobei er durch das ***EFFECTS***-Framework weitesgehend untersützt wird.

KAPITEL 19

Die Systemarchitektur

Im Folgenden wird die Systemarchitektur des ***EFFECTS***-Plugins beschrieben. Alle hier aufgeführten Komponenten und Klassen sind in dem Java Paket `de.ls10.effects.core` zu finden. Abbildung 19.1 zeigt einen Überblick über das Paket. Betrachtet man zunächst ganz allgemein das System, so lässt es sich grob in zwei Teile gliedern:

1. Die *Kernkomponenten* bilden die Basis des Plugins. Diese Komponenten werden benötigt, um Diagramme darzustellen, zu speichern und zu laden. Des Weiteren ermöglichen sie die Navigation durch die dreidimensionale Welt und die Interaktion mit Objekten.

2. Die *Hilfskomponenten* kapseln häufig benötigte Funktionen, welche z.B. die Nutzung des *Eclipse*-internen Datenmodells, sowie die Handhabung von Java3D erleichtern.

In dem nächsten Abschnitt wird näher auf die Kernkomponenten eingegangen. Es wird beschrieben, aus welchen Paketen diese bestehen und wozu sie im einzelnen dienen. Das allgemeine Konzept des Systems wird dabei erläutert. Anschließend werden die Klassen innerhalb der Pakete beschrieben und deren Zusammenarbeit dargestellt. Nach Beschreibung der Kernkomponenten, werden dann die Hilfskomponenten näher erläutert. Abschließend wird das Gesamtsystem und sein Ablauf beschrieben.

19.1 Die Kernkomponenten

Michél Kersjes, Christian Mocek

Als Kernkomponenten werden die Teile des ***EFFECTS***-Plugins bezeichnet, welche nötig sind, die Basisfunktionalität anzubieten. Ohne diese ist es nicht möglich, das Framework als solches zu nutzen, eigene Diagrammtypen zu entwickeln und in *Eclipse* darzustellen. Die Kernkomponenten sind in Abbildung 19.2 dargestellt und befinden sich in folgenden Paketen:

- Das Paket `datamodel` realisiert durch seine Klassen die Verwaltung eines eigenen Datenmodells, welches das *Eclipse*-interne Datenmodell erweitert.

- Die Aufgabe des Pakets `datamodel.properties` ist die Visualisierung des eigenen Datenmodells in dem *Eclipse*-Property-View.

- Der Editor wird im gleichnamigen Paket `editor` realisiert und dient der Darstellung und Manipulation des Diagramms durch den Anwender.

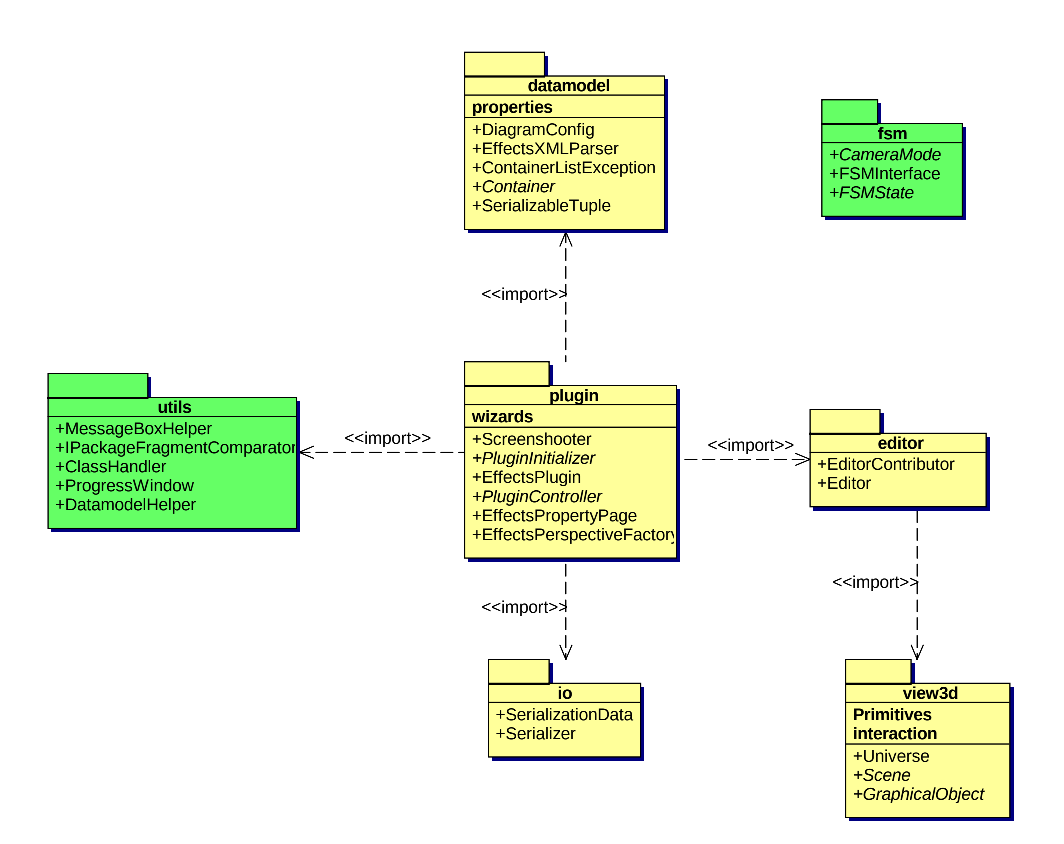


Abbildung 19.1.: Die Kernkomponenten im Überblick

- Das Diagramm persistent zu speichern und zu laden, ermöglichen die Klassen des Pakets `io`.
- Die Schnittstelle zwischen *Eclipse* und dem Framework wird im Paket `plugin` beschrieben. Zusätzlich wird hier die ***EFFECTS***-Perspektive erzeugt.
- Das Paket `plugin.wizards` hat die Aufgabe, die Implementierung eines Wizards zum Anlegen eines neuen Diagramms zu ermöglichen.
- Die Realisierung der 3D-Szene ist im Paket `view3d` gekapselt.
- Die Navigation im dreidimensionalen Raum wird durch eine Kombination aus Maus und Tastatursteuerung realisiert, welche im Paket `view3d.interaction.navigation` bereitgestellt wird.
- Um mit Objekten zu interagieren, ist das Auswählen dieser in der 3D-Szene notwendig. Die Funktionalität hierzu ist im Paket `view3d.interaction.picking` zu finden.

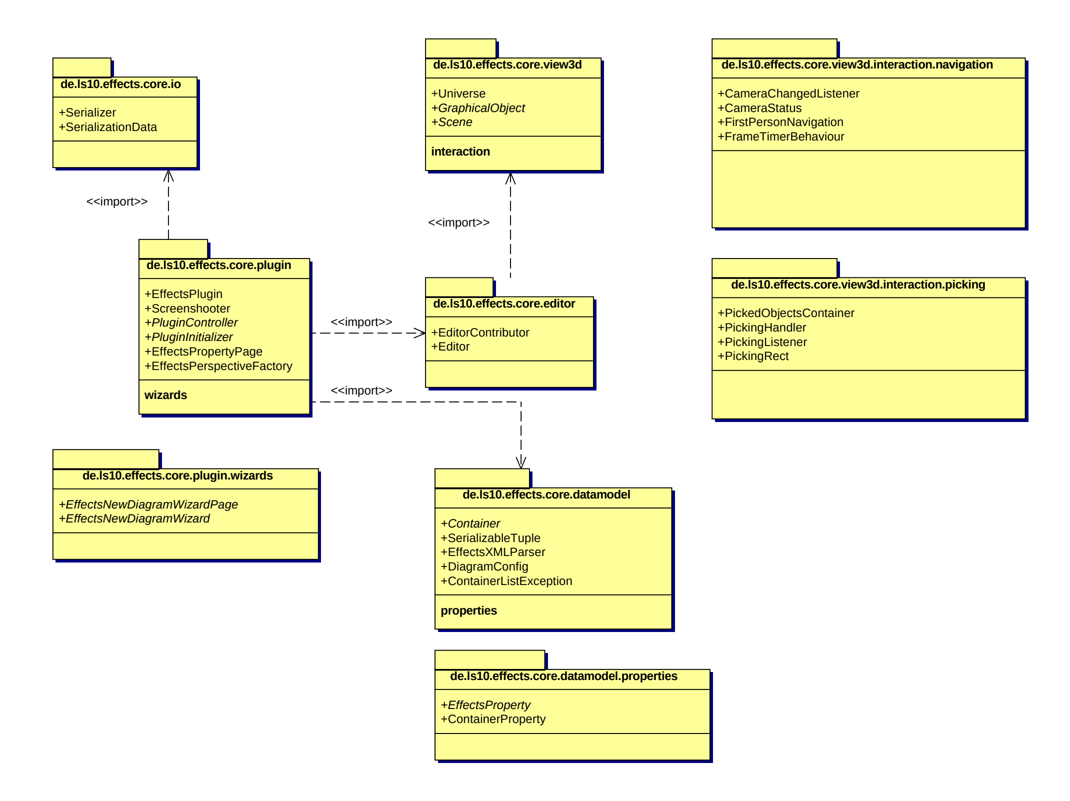


Abbildung 19.2.: Die Kernkomponenten des Frameworks

Um den Zusammenhang der oben aufgeführten Pakete erläutern zu können, ist zunächst zu erwähnen, nach welchem Konzept das Framework entwickelt wurde. Um eine möglichst hohe

Flexibilität zu erreichen, wird das allgemein anerkannte Konzept des *Model-View-Controllers* verwendet. Die Wahl dieses Konzepts wurde deshalb durchgeführt, da es den Anforderungen an das Plugin, ein Framework zur einfachen Erstellung von Diagrammen zur Visualisierung und Modifizierung von Softwarestrukturen, entgegenkommt. Den Modell-Teil findet man in einem Datenmodell, welches durch Informationen aus dem Java-Sourcecode gefüllt wird. Dieses Datenmodell wird von *Eclipse* zur Verfügung gestellt und zwar als Teil des *Java Development Toolkits (JDT)*. Es wird über die Controller des ***EFFECTS***-Plug-Ins modifiziert und als View in Form von einer dreidimensionalen Szene dargestellt. Die Flexibilität wird dadurch erreicht, dass es durch das Framework möglich ist, die einzelnen Teile Model, View und Controller frei zu spezifizieren.

Nachdem das verwendete Konzept erläutert ist, kann der Zusammenhang der Pakete dargestellt werden. Zu erkennen ist, dass sich die Teile Modell, View und Controller in der Paketstruktur widerspiegeln. Zentrale Punkte sind die Pakete `plugin` und `editor`. Diese bilden die Controller- Komponente des MVC. Der Modell-Teil wird über das Paket `datamodel` abgebildet, während der View-Teil über das Paket `view3d` realisiert wird.

Im Gegensatz zu den oben genannten Paketen, besitzen die folgenden keinen direkten Bezug zum MVC-Konzept. Die Funktionalität der Pakete `view3d.interaction.picking` und `view3d.interaction.navigation` wird von dem Paket `view3d` verwendet. Ähnliches gilt hierbei auch für die Pakete `plugin.wizards` und `io`. Sie werden von der Controller-Komponente verwendet, um neue Diagramme anlegen zu können und diese zu laden und zu speichern.

19.1.1 Die Pakete im Detail

In diesem Absatz wird darauf eingegangen, wie die oben genannten Aufgaben der einzelnen Pakete durch die ihnen zugrunde liegenden Klassen realisiert werden.

Verwaltung eines eigenen Datenmodells

Das Paket `datamodel` hat die oben genannte Aufgabe für einen Diagrammtyp ein spezialisiertes Datenmodell zu verwalten. Hierzu sind die folgenden Klassen implementiert:

- Die Klasse `Container` bildet die Basis zur Erweiterung des *Eclipse*-internen Datenmodells. Um diese durchzuführen, wird an eine Ressource in dem Arbeitsbereich von *Eclipse* ein Objekt dieses Containers angehängt. Eine Ressource wird in *Eclipse* als `IResource` dargestellt. Eine `IResource` ist eine Modellklasse, die entweder ein Projekt, ein Verzeichnis, eine Datei oder den Vaterknoten aller Projekte repräsentiert. Sollte bei der Verwaltung der Container ein Fehler auftreten, so wird eine Ausnahme vom Typ `ContainerListException` ausgelöst.

- Die Zuordnung von Containertyp zu einer `IResource` wird in einem Objekt der Klasse `DiagramConfig` gespeichert.

- Der `EffectsXMLParser` liest eine zuvor angelegte XML-Datei ein, in welcher die oben genannte Zuordnung persistent gespeichert wird.

Der Zusammenhang zwischen diesen Klassen ist aufgrund der Funktionalität der Einzelklassen ersichtlich. Der XML-Parser liest die Zuordnungen aus der XML- Datei. Anschlie-

ßend werden die entsprechenden Container-Objekte angelegt und an die Ressource gebunden. Außerdem werden die Zuordnungen in dem Objekt der Klasse `DiagramConfig` gespeichert.

Visualisieren des Datenmodells im Property-View

Um Eigenschaften eines beliebigen Objektes ändern zu können, bietet *Eclipse* die Möglichkeit den so genannten *Property-View* zu verwenden. Diese Möglichkeit wird genutzt, um Werte des Datenmodells anzuzeigen und ggf. zu ändern. Um dies zu realisieren, existieren die folgenden Klassen:

- Um in einem Diagramm spezifische Daten im Property-View verwenden zu können, existiert die abstrakte Klasse `EffectsProperty`.
- Die Klasse `ContainerProperty` ermöglicht es, Änderungen an einem Container des Datenmodells durchzuführen. Sie erweitert die Klasse `EffectsProperty`.

Realisierung des Editors

Um eine Darstellung der Szene und die mögliche Interaktion überhaupt zu ermöglichen, wird ein *Eclipse*-Editor zur Verfügung gestellt. Die Klasse, welche hierzu existiert ist `Editor`. Er verarbeitet das vom ***EFFECTS***-Plugin vorgegebene Dateiformat *efx*. Innerhalb von Dateien dieses Formats werden Diagramme persistent gespeichert. Im Zusammenhang mit der Navigation erkennt der Editor, wenn der Anwender die Kamera im dreidimensionalen Raum bewegt.

Speichern und Laden von Diagrammen

Die im Paket `io` verfügbaren Klassen realisieren das Speichern und Laden von Diagrammen in eine Datei. Hierzu existieren die beiden folgenden Klassen:

- Objekte der Klasse `SerializationData` beschreiben die Daten eines Diagramms, die in eine Datei des Formats *efx* gespeichert werden sollen. Diese Daten werden in einem Diagramm spezifiziert.
- Um die spezifizierten Daten zu speichern und zu laden, existiert die Klasse `Serializer`.

Anlegen eines neuen Diagramms

Das Paket `plugin` stellt die Klassen zur Verfügung, welche die Schnittstelle zwischen dem Plugin und *Eclipse* darstellen. In diesem Paket befinden sich auch die Klassen, die die Controller-Komponente des MVC-Konzepts bilden. Die Aufgaben der einzelnen Klassen werden im Folgenden beschrieben:

- Die Klasse `PluginController` ist die Basisklasse des Plugins und stellt die Schnittstelle zwischen *Eclipse* und dem Plugin dar. Sie bietet Methoden zur Verwaltung des eigenen Datenmodells und Methoden um auf den Editor zuzugreifen.
- Die Initialisierung des eigenen Datenmodells, sowie das Registrieren der vom Plugin benötigten Klassen beim Controller, geschieht mit Hilfe einer Instanz von der Klasse `PluginInitializer`. Ausserdem wird von einem Objekt dieser Klasse eine XML-Konfigurationsdatei eingelesen, in der spezifiziert ist, wie das *Eclipse*-Datenmodell um das eigene zu erweitern ist.

- Das Objekt der Klasse `EffectsPlugin` wird von der *Eclipse*-Workbench zur Kommunikation mit dem Plugin benötigt.
- Um Daten zu einem Objekt aus der 3D-Szene im *Property-View* von *Eclipse* anzuzeigen, existiert die Klasse `EffectsPropertyPage`. Eine Instanz dieser Klasse reagiert, sobald in der 3D-Szene ein Objekt ausgewählt wird und stellt dessen Daten der Instanz der Klasse `EffectsProperty` zur Verfügung.
- Die Klasse `EffectsPerspectiveFactory` hat die Aufgabe das Layout der *Eclipse*-Perspektive des Plugins festzulegen.
- Mit Hilfe der Klasse `Screenshooter` kann ein Screenshot der 3D-Szene erstellt werden.

Ein direkter Zusammenhang besteht zwischen der Klasse `PluginController` und der Klasse `PluginInitializer`. Letzterer erzeugt nach dem registrieren der vom Controller benötigten Klassen sowie dem Einlesen der Konfigurationsdatei den `PluginController`, welcher die eigentliche Kontrollkomonente im Sinne des MVC-Konzepts darstellt.

Anlegen eines neuen Diagramms mit einem Wizard

Wenn ein neues Diagramm erzeugt werden soll, so wird für dieses Diagramm eine Datei mit der Endung *efx* benötigt. Diese Datei wird mit Hilfe von Wizards erzeugt. Um solche Wizards anzulegen gibt es folgende Klassen im Paket `plugin.wizards`:

- Die Klasse `EffectsNewDiagramWizardPage` stellt eine Seite für den Wizard zur Verfügung. Über solch eine Seite wird der Container für die *efx*-Datei ausgewählt, sowie der Dateiname für die Datei festgelegt. Ein Container ist eine Ressource von *Eclipse*, wie zum Beispiel ein Projekt.
- Die *efx*-Datei wird mittels eines Objekts der Klasse `EffectsNewDiagramWizard` erzeugt. Dem Wizard wird ein Objekt der Klasse `EffectsNewDiagramWizardPage` hinzugefügt, über welches der eigentliche Wizard die benötigten Daten zur Erstellung der *efx*-Datei erhält.

Realisierung und Darstellung der 3D Szene

Die View-Komponente des MVC-Konzepts wird durch die Klassen des Pakets `view3d` realisiert. Es existieren drei Klassen, deren Aufgaben im Folgenden beschrieben werden:

- Die Klasse `Universe` stellt eine Java3D-Umgebung zur Verfügung, in welcher die diagrammspezifische Szene gezeichnet wird. Eine Szene ist hierbei die grafische Darstellung von Strukturen oder Objekten, wie zum Beispiel Klassen- oder Sequenzdiagrammen.
- Da die Art und Anordnung grafischer Objekte in einer 3D-Szene diagrammspezifisch ist, wird eine Schnittstelle zwischen der allgemeinen View-Komponente, dem Universum, und der in einem Diagramm generierten Szenenbeschreibung benötigt. Diese Aufgabe erfüllt die abstrakte Klasse `Scene`, indem sie abstrakte Methoden zur Verfügung

stellt. Die Controller-Komponente kennt jeweils die konkrete Implementierung der Szene und nutzt die Methoden, um dem Universum, also der View-Komponente, die darzustellende Szene zu übergeben.

- Um die Flexibilität zu gewährleisten, dass 3D-Objekte in verschiedenen Diagrammen wiederverwendet werden können und austauschbar sind, existiert die abstrakte Klasse `GraphicalObjects`. Sie stellt Methoden zur Verfügung um zum Beispiel das Objekt direkt im 3D-Raum zu positionieren oder die Transparenz des Objekts einzustellen.

Navigation im dreidimensionalen Raum

Das Framework stellt eine aus vielen 3D-Spielen bekannte Kombination aus Maus- und Tastatursteuerung zur Verfügung. Die Mausbewegung führt zu einer Rotation der Kamera, während Eingaben auf der Tastatur zu einer Kamerabewegung führen. Diese Navigation ist im Paket `view3d.interaction.navigation` realisiert. Die folgenden Klassen implementieren die beschriebene Funktionalität:

- Die Klasse `FrameTimerBehavior` realisiert einen Java3D-Behavior. Dieser ist so konfiguriert, dass bei jedem Neuzeichnen eines Bildes der Szene die neue Kameraposition und -blickrichtung berechnet wird. Die Kamera wird nach der Berechnung an die entsprechende Position verschoben und gedreht.

- Da der `FrameTimerBehavior` nur für genau ein Bild die nötige Kamerabewegung berechnet, ist eine Klasse notwendig, die auf Tastatureingaben durch den Benutzer bzw. auf die von der Interaktion mit der Maus erzeugten Ereignisse reagiert. Diese Verarbeitung wird durch die Klasse `FirstPersonNavigation` erledigt, welche vom `FrameTimerBehavior` erbt. Die empfangenen und verarbeiteten Ereignisse werden dazu genutzt, den Behavior zu steuern. So werden zum Beispiel Mausbewegungen in Winkel umgerechnet und diese an den Behavior übergeben, so dass die Kamera um den berechneten Winkel rotiert wird.

Um zusätzlich eine Möglichkeit zu schaffen, anderen Objekten Änderungen der Kamera mitzuteilen, wurde das Interface `CameraChangedListener` geschaffen. Objekte welche dieses Interface implementieren und sich als Listener am `FrameTimerBehavior` oder respektive der erbenden Klassen registrieren, werden über Kameraänderungen informiert und erhalten ein Objekt vom Typ `CameraStatus`. Diese Klasse enthält Informationen über die Blickrichtung der Kamera und deren Position im 3D-Raum zu dem Zeitpunkt als das `CameraStatus` Objekt erzeugt wurde.

Auswählen von Objekten im Raum

Das letzte Paket `view3d.interaction.picking` der Kernkomponenten dient zur Interaktion mit grafischen Objekten und ermöglicht das anklicken dieser Objekte in einer Szene. Folgende Klassen sind sind hierzu erforderlich:

- Ein Objekt der Klasse `PickingHandler` reagiert auf Mausklicks in der 3D-Darstellung und ermittelt die angeklickten grafischen Objekte. Zusätzlich haben beliebige andere Klassen bzw. deren Objekte die Möglichkeit, sich ihrerseits beim `PickingHandler`

zu registrieren um darüber informiert zu werden, wenn grafische Objekte angewählt wurden. Diese Klassen müssen das `PickingListener` Interface implementieren und erhalten dann eine Liste der angewählten 3D-Objekte.

- Die angeklickten Objekte werden vom `PickingHandler` in einem Container verwaltet. Dieser Container wird durch die Klasse `PickedObjectsContainer` realisiert.

Durch die Verwendung eines Containers zur Verwaltung der angeklickten Objekte ist es möglich, zu jedem beliebigen Zeitpunkt die aktuelle Auswahl abzufragen. Des Weiteren ist es so möglich, die ausgewählten Objekte in der 3D-Darstellung zu markieren, was durch einen Rahmen geschieht. Dieser wird durch die Klasse `HighlightBox` zur Verfügung gestellt. Wird also ein Objekt in den Container eingefügt, so wird dem Anwender durch die automatische Markierung des Objekts in der Darstellung angezeigt, dass dieses Objekt ausgewählt ist. Wird ein Objekt aus dem Container entfernt, so wird auch die Markierung in der Darstellung entfernt. Aus diesen Gründen wurde darauf verzichtet bei einer Auswahl von grafischen Objekten in der 3D-Darstellung ausschließlich alle Listener zu informieren und dann die durchgeführte Auswahl zu verwerfen.

19.1.2 Das Zusammenspiel der Komponenten

Kai Gutberlet, Jan Wessling

Abbildung 19.3 zeigt eine Übersicht der wichtigsten Komponenten, geordnet nach dem MVC-Konzept. Die Klassen `Editor`, `PluginInitializer` und `PluginController` stellen die Controller-Komponente dar. Dabei wird der `PluginInitializer` nur dazu verwendet das Plugin zu initialisieren, während der `PluginController` die zentrale Komponente des Systems ist. Er verwaltet das Datenmodell, indem er dafür sorgt, dass `Container` an die entsprechenden Ressourcen des *Eclipse*-Datenmodells gebunden werden. Über den Editor steuert der `PluginController` den View. Der Editor selbst verwaltet das Universum und die darzustellende Szene, welche durch die Klassen `Universe` und `Scene` realisiert sind.

Das Zusammenspiel dieser Klassen soll jetzt näher erläutert werden. Dazu zeigt Abbildung 19.4 den Ablauf beim Öffnen einer *efx* Datei. Das Öffnen bewirkt eine Instanziierung der Klasse `Editor`, welche dann ein Objekt vom Typ `PluginInitializer` erzeugt. Dessen Aufgabe die Initialisierung des ***EFFECTS***-Plugins. Hierbei wird ein `EffectsXMLParser` erzeugt, der die Konfigurationsdaten des Diagrammtyps aus einer XML-Datei ausliest und zurückgibt. Danach wird der diagrammspezifische `PluginController` erzeugt, der nach Aufruf von `setEditor` den weiteren Ablauf je nach Diagrammtyp steuert. Dieser ist am Beispiel vom ClassPackageDiagramm aus Release 4 in Abbildung 19.5 dargestellt.

Der `CPDPluginController` erzeugt bei seiner Instantiierung eine `DataModelFacade`, die das Datenmodell kapselt und zusätzliche Methoden auf dem Datenmodell bereitstellt. Des Weiteren wird der Zustandsautomat des Diagramms instanziiert. Sobald der `Editor`, wie schon in Abbildung 19.4 beschrieben, die Methode `setEditor` aufruft, wird die diagrammspezifische Darstellung der `Scene` neu berechnet. Dazu ruft der `CPDPluginController` die eigene Methode `updateEditors` auf, welche dann eine neue `Scene` erstellt, den Startzustand im Zustandsautomaten speichert und `updateView` aufruft. Dort wird im `Editor` mittels

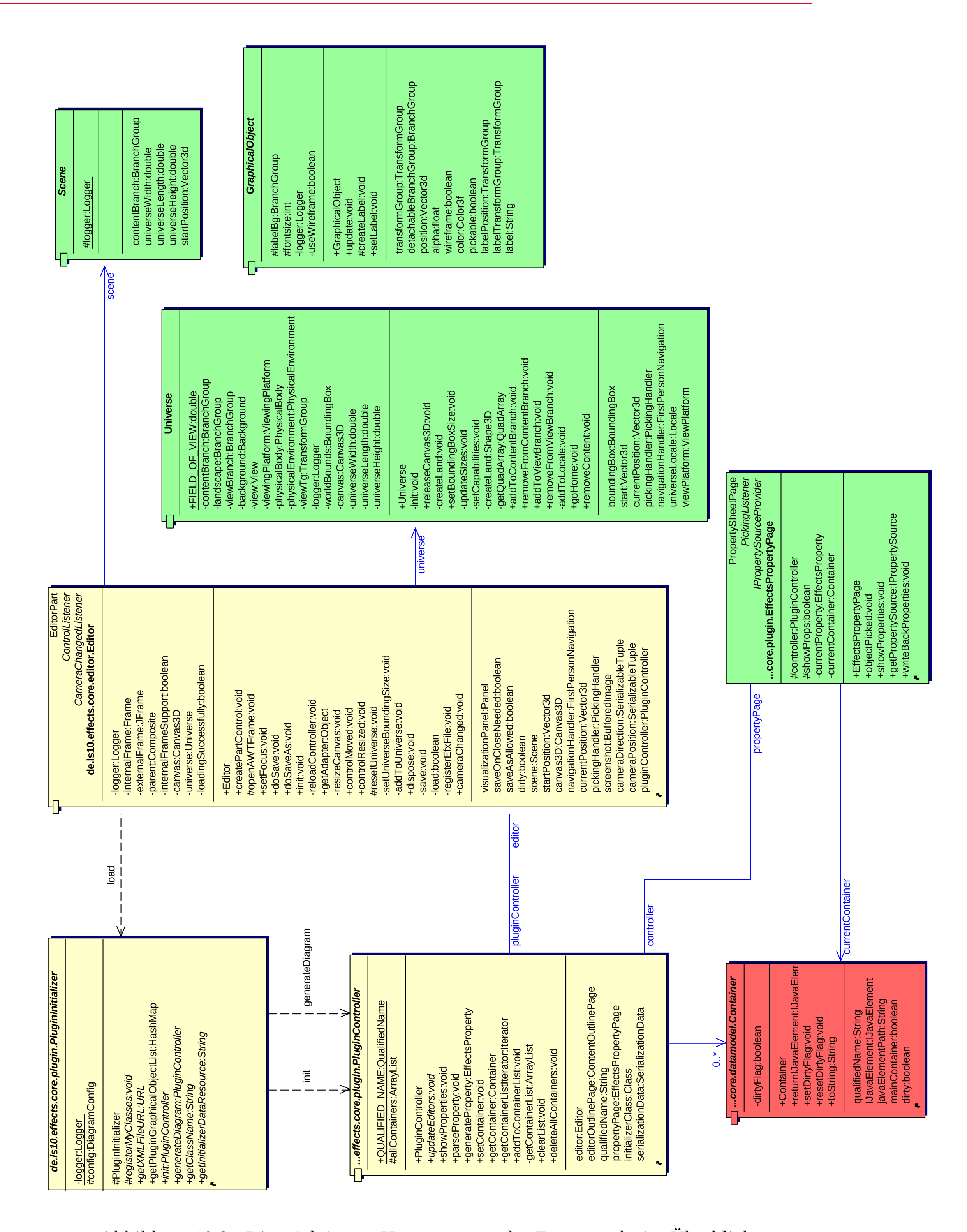


Abbildung 19.3.: Die wichtigsten Komponenten des Frameworks im Überblick

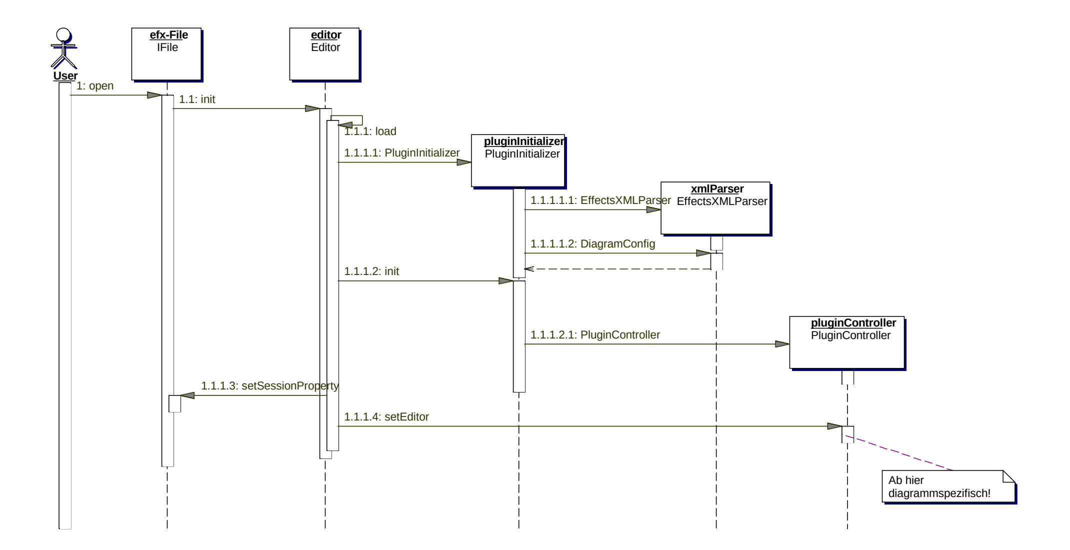


Abbildung 19.4.: Ablauf beim Öffnen einer *efx* Datei

`setScene` die `Scene` neu gesetzt. Dieser Aufruf erfolgt nach jedem Neuzeichnen der `Scene` und führt dazu, dass sich die Anzeige im Editorfenster gemäß des neuen Inhalts aktualisiert.

Der weitere Ablauf hängt ab hier nur noch von Benutzereingaben ab. Das Reagieren auf Benutzereingaben muss natürlich diagrammspezifisch erfolgen. Hierzu kann man einen Zustandsautomaten verwenden.

19.2 Die Hilfskomponenten

Christian Mocek, Michael Pflug, Daniel Unger

Die Hilfskomponenten stellen häufig gebrauchte Funktionen bereit, sowie Funktionen, die den Umgang mit *Eclipse*-internen Klassen erleichtern. Sie werden in Abbildung 19.6 dargestellt. Folgende Komponenten können hierbei unterschieden werden:

- Das Paket `fsm` enthält alle notwendigen Klassen, um einen konkreten Zustandsautomaten zu erstellen, der dann in einem Plug-In-Fragment, das die jeweilige vom Benutzer gewünschte Funktionalität darstellt, genutzt werden kann. Ein Zustandsautomat kann genutzt werden, wenn zwischen verschiedenen Modi wie zum Beispiel einem Navigations- und einem Editiermodus gewechselt werden muss. Jeder Modus wird als Zustand implementiert und erlaubt dann bestimmte Folgezustände, die dann eine Folgeaktion sind.
- Um eine 3D-Szene mit „Leben“ zu füllen, müssen grafische Objekte vorhanden sein. Grundlegende geometrische Figuren wie Boxen, Kugeln und Pfeile bietet das Paket

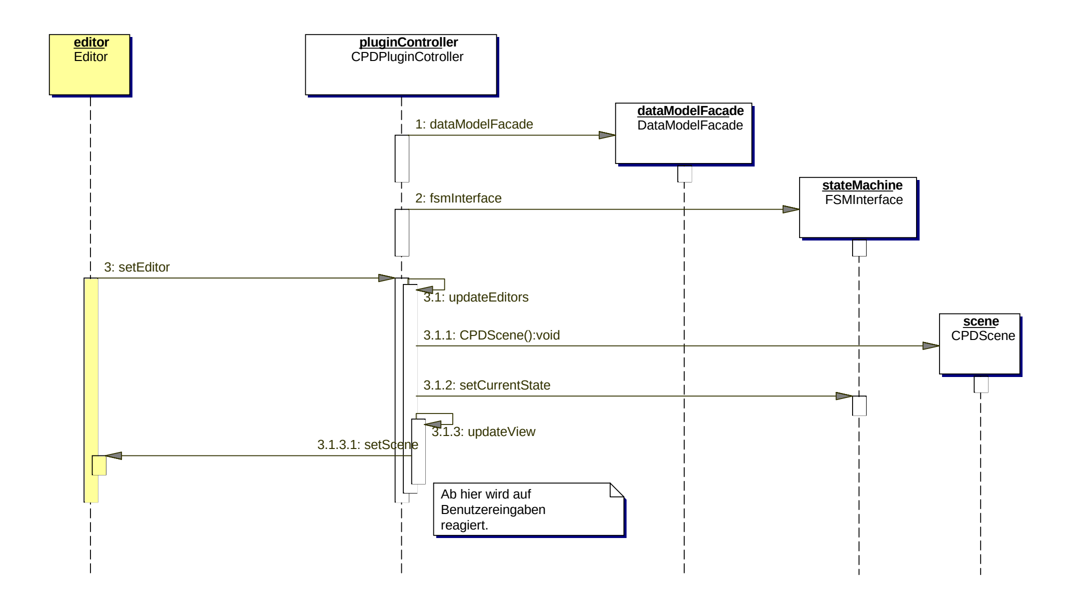


Abbildung 19.5.: Beispielhafter Ablauf beim Initialisieren des PluginControllers

`view3d.primitives`. Alle in diesem Paket vorhandenen Primitive erben von der abstrakten Klasse `GraphicalObjects` und können so in verschiedenen Diagrammen eingesetzt werden.

- Das Paket `view3d.interaction.objecttranslation` ermöglicht das Verschieden der Objekte im dreidimensionalen Raum auf Basis von Transaktionen, wie sie aus dem Bereich der Datenbanken bekannt sind.
- Um immer wieder benötigte Funktionen bereitzustellen, wurden Hilfsklassen geschrieben, die in dem Paket `utils` liegen.

Alle oben aufgeführten Pakete sind als eigenständige Komponenten zu sehen. Jede Komponente bzw. jedes Paket bietet seine eigene Funktionalität, welche unabhängig von den anderen Paketen ist, so dass ein Zusammenhang zwischen diesen Komponenten nicht gegeben ist.

19.2.1 Die einzelnen Hilfskomponenten im Detail

Nachfolgend wird auf die Klassen und Interfaces der einzelnen Pakete eingegangen. Dabei soll der Verwendungszweck der einzelnen Klassen erörtert werden, so dass die Bedeutung dieser Klassen deutlich wird.

Realisierung von Zustandsautomaten

Das Paket `fsm` bietet ein Entwurfsmuster zur Erstellung von endlichen Zustandsautomaten an. Zunächst werden die beiden dafür benötigten Klassen erläutert:

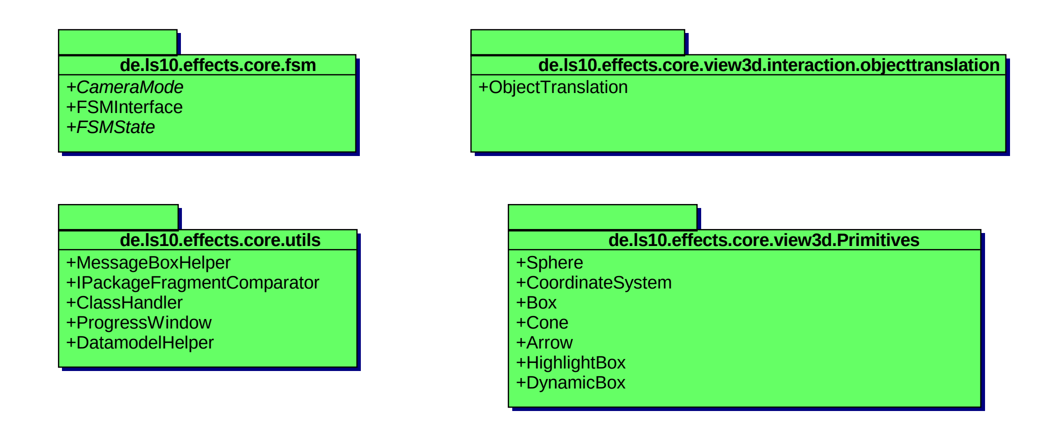


Abbildung 19.6.: Die Hilfskomponenten des Frameworks

- Das Interface `FSMInterface` dient dazu, auf die auftretenden Ereignisse in einem Automaten zu reagieren. Mit Hilfe dieses Interfaces können alle Zugriffe auf die Zustände des implementierten Automaten realisiert werden, ohne dass die jeweiligen Zustände konkret bekannt sein müssen.

- Die abstrakte Klasse `FSMState` ist die Basisklasse für die verschiedenen Zustände in einem Automaten. Jeder Zustand in einem speziellen Automaten muss von dieser Klasse erben und die entsprechenden Methoden müssen dann je nach Kontext entsprechend implementiert werden.

- Die Klasse `CameraMode` stellt den Standardzustand eines Automaten bereit. Dieser beinhaltet keine besondere Funktionalität, sondern dient lediglich als der Zustand, von dem alle anderen Folgezustände ausgehen.

Die Verwendung eines Zustandsautomaten ist im Detail in Kapitel 17.5.1 erläutert.

Grundlegende grafische Objekte

Das Paket `view3d.primitives` stellt verschiedene Primitive an grafischen Objekten bereit, welche in verschiedenen Diagrammen genutzt werden können (siehe Abbildung 19.7).

- Die Klasse `Arrow` stellt einen dreidimensionalen Pfeil dar. Dabei kann Anhand der verschiedenen Konstruktoren entschieden werden, ob zum Beispiel ein Pfeil mit oder ohne Spitze erzeugt wird.

- Die Klasse `Box` stellt eine in Farbe und Größe frei einstellbare Box dar.

- Das grafische Objekt `Cone` repräsentiert einen Kegel, wie er bereits von der Java3D-API zur Verfügung gestellt wird. In Diagrammen sollte allerdings nur dieser Kegel Verwendung finden, da dieser von `GraphicalObject` erbt und nur so die Interaktion, also das Anwählen und Verschieben funktioniert. Dies ist mit dem Java3D-Kegel nicht möglich.

- Mittels der Klasse CoordinateSystem wird ein Koordinatensystem visualisiert, mit dessen Hilfe man die Achsen der Objekte darstellen kann.
- Mit der Klasse DynamicBox wird eine Box bereitgestellt, welche nicht in ihrer Größe beeinflusst werden kann, sondern die sich in der Größe automatisch so anpasst, dass all in ihr vorhandenen Elemente von der Box umschlossen werden.
- Die Klasse HighlightBox erzeugt einen Rahmen um beliebige grafische Objekte. Dies wird zum Beispiel zur Markierung der selektierten Objekte verwendet. Die Größe des Rahmens wird automatisch berechnet.
- Ähnlich wie beim Kegel, gilt auch für Kugeln das gleiche Problem mit der von Java3D vorgegebenen Primitiven. Das Paket Sphere realisiert eine Kugel, welche ebenfalls von GraphicalObject erbt und zur Interaktion geeignet ist.

Verschieben von Objekten

Das Paket view3d.interaction.objecttranslation beinhaltet die zum Verschieben von grafischen Elementen in der 3D-Szene benötigte Klasse ObjectTranslation Hierzu werden die zu verschiebenden Objekte der Instanz der Klasse bekannt gemacht und eine Transaktion gestartet, welche bestätigt oder abgebrochen werden muss. Aus den Mausbewegungen wird ein Richtungsvektor berechnet, um den die Objekte verschoben werden.

Sonstige Hilfsklassen

Das Paket utils enthält einfache Hilfsklassen, die im Folgenden erläutert werden.

- Da Klassen aus einem Plug-In Fragment nicht direkt in das Core-Plug-In geladen werden können, bietet die Klasse ClassHandler Möglichkeiten an, Klassen über den entsprechenden Klassennamen einmalig per Reflection-Mechanismus zu laden und diese dann in dem Core-Plug-In zur Verfügung zu stellen.
- Die Klasse DatamodellHelper bietet Funktionen an, um auf das *Eclipse*-Datenmodell zuzugreifen. So können beispielsweise alle direkten Unterpakete eines ausgewählten Pakets abgefragt werden oder Pakete entfernt werden, die leer sind.
- Der IPackageFragmentComparator implementiert das Java Comperator Interface und ermöglicht den Vergleich zweier IPackageFramgent-Namen, welche die Repräsentation eines Java-Pakets im *Java Development Toolkit* sind.
- Die Klasse MessageBoxHelper vereinfacht die Verwendung von Infofenstern. Es werden verschiedene statische Methoden angeboten, die das Infofenster automatisch so einstellen, dass nur noch die anzuzeigende Nachricht angegeben werden muss. Zum Beispiel erzeugt die Methode zur Anzeige einer Frage ein Infofenster mit einem Fragezeichen als Icon und einem Ja bzw. Nein Knopf.
- Das Framework enthält einen Fortschrittsdialog, welcher in der Klasse ProgressWindow realisiert ist. Dieser Dialog kann neben dem Fortschrittsbalken zusätzliche Nachrichten anzeigen über den aktuellen Vorgang anzeigen.

–

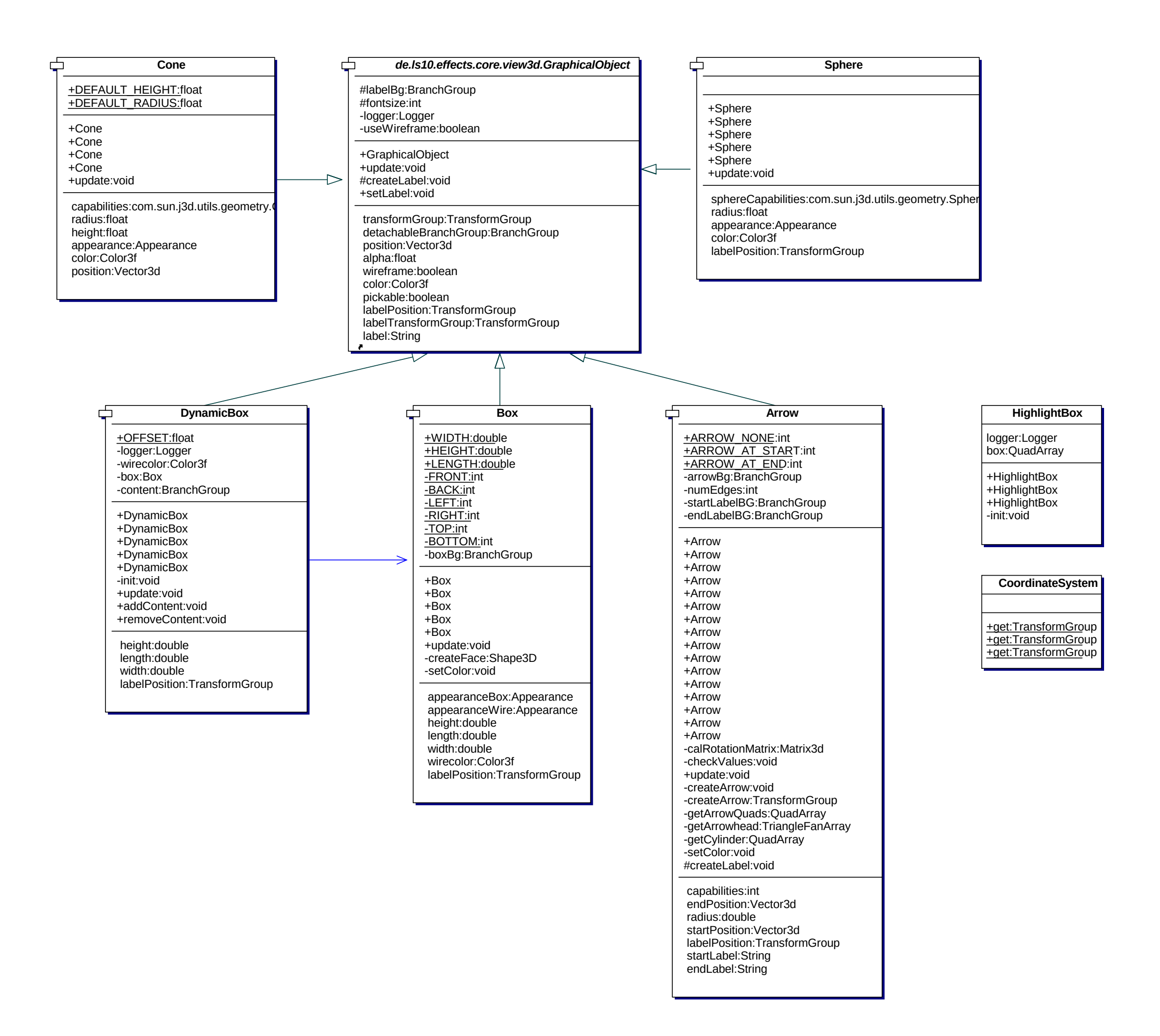


Abbildung 19.7.: Übersicht der grafischen Primitive

KAPITEL 20

Vorgehensweise zum Erstellen eines neuen Diagrammtyps

Michél Kersjes, André Kupetz, Christian Mocek, Sven Wenzel

In dieser Anleitung soll das grundlegende Vorgehen erläutert werden, um ein eigenes auf dem ***EFFECTS***-Framework basierendes Plug-In Fragment zu erstellen. Es wird vorausgesetzt, dass das ***EFFECTS***-Core-Plug-In korrekt installiert ist.

20.1 Anlegen des Projektes

1. Zunächst muss in *Eclipse* ein neues Projekt vom Typ `Plug-in Fragment` erstellt werden (siehe Abbildung 20.1). Der angegebene Projektname wird automatisch als Bezeichner für das Fragment gewählt.

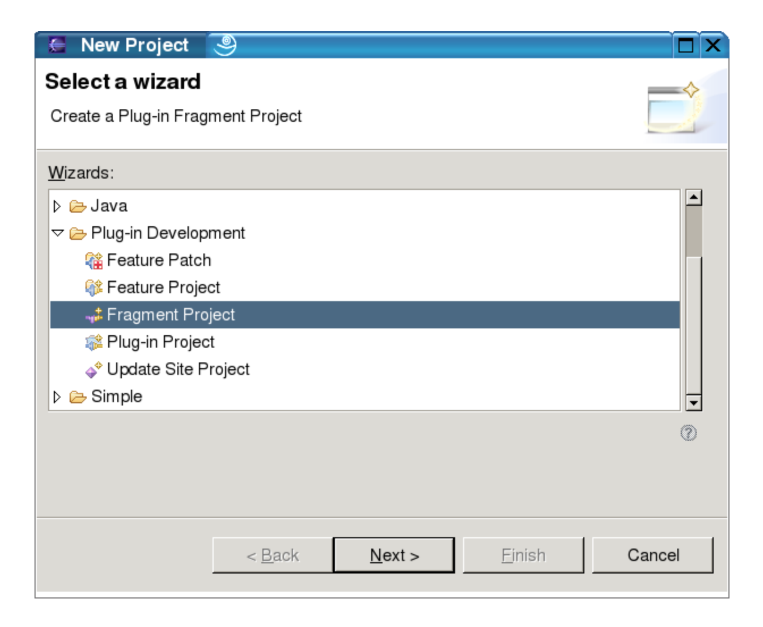


Abbildung 20.1.: Neues Projekt erstellen

2. In dem nachfolgenden Dialog (siehe Abbildung 20.2) muss nun das ***EFFECTS***-Plug-In als Parent gewählt werden. Hierzu wird die Plug-in ID `de.ls10.effects.core` angegeben. Alle weiteren Werte in diesem Dialog können optional geändert werden.

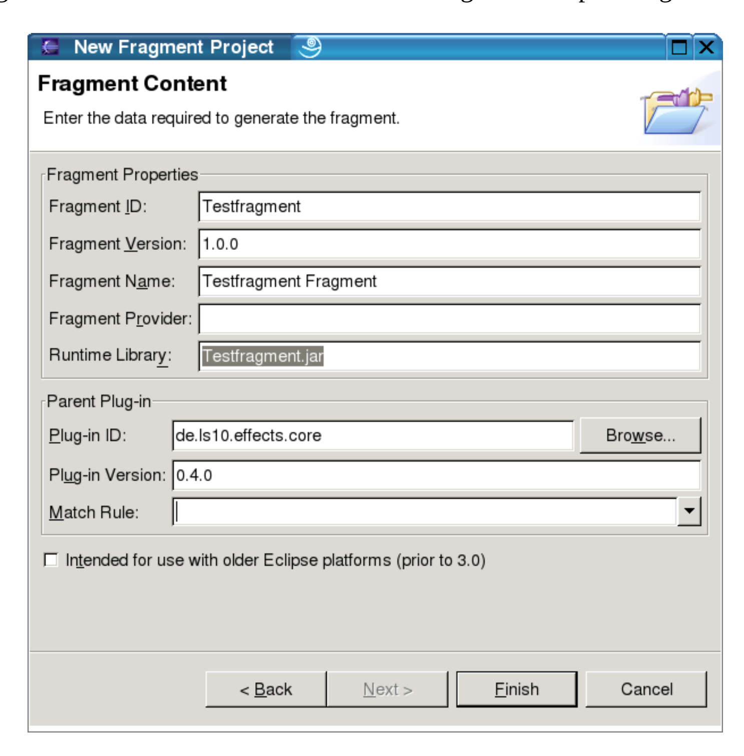


Abbildung 20.2.: Basisplugin festlegen

20.2 Hinzufügen von Fragment Extensions

Der nächste Schritt ist das Erweitern der *Eclipse* Plattform durch den Extension-Mechanismus. Dies geschieht in der `fragment.xml` des neu angelegten Plugins. Zwingend notwendige Erweiterungen sind:

- `org.eclipse.ui.newWizards` um einen Wizard zu implementieren, der dafür sorgt, das neue Diagramme angelegt werden. Ein ***EFFECTS***-Wizard muss in der Kategorie `de.ls10.effects.core.wizards` registriert sein.

- `de.ls10.effects.core.DiagramGeneration`, da hierüber die Klassen des Controllers und der Szeneberechnung dem Basisplugin bekannt gemacht werden.

Optional können beliebige weitere Extensions zur *Eclipse*-Plattform hinzugefügt werden. Dies könnten zum Beispiel Menüerweiterungen und `PerspectiveExtensions` sein, um die

EFFECTS-Perspektive zu erweitern. Für nähere Informationen wird an dieser Stelle auf die *Eclipse*-interne Hilfe verwiesen.

20.3 Implementierung wichtiger Klassen

In diesem Abschnitt wird näher auf das Zusammenspiel der einzelnen Klassen eingegangen und die wichtigen Schritte erläutert, die durchgeführt werden müssen, um ein neues Diagramm lauffähig zu machen.

20.3.1 Der neue Eclipse Wizard

Paket: `de.ls10.effects.core.plugin.wizards`

Wie oben erwähnt, ist es für jedes Diagramm notwendig einen eigenen Wizard zu implementieren. Hierzu müssen mindestens zwei Klassen angelegt werden. Zum ersten der eigentliche Wizard, welcher von der Klasse `EffectsNewDiagramWizard` erbt und folgende Aufgaben erfüllen muss:

- Erzeugen und Registrieren der einzelnen Wizard-Seiten
- Festlegen des Diagramm-Namens
- Festlegen der `SerializationData`, die Beschreiben, wie das Diagramm persistent gespeichert werden kann

Die zweite anzulegende Klasse erbt von der Klasse `EffectsNewDiagramWizardPage`. Mit Hilfe dieser Klasse wird das Layout des Wizards spezifiziert.

Die einzelnen Seiten müssen von der Klasse `EffectsNewDiagramWizardPage` abgeleitet werden und stellen das Aussehen der Seiten und Erfassen der Eingabedaten bereit. Diese Daten müssen vom eigentlichen Wizard ausgelesen werden können, damit dieser die Informationen dazu verwenden kann, die benötigten Aktionen durchzuführen. Diese Aktionen sind i.d.R. diagrammspezifisch und bestehen zum Beispiel im Vorbelegen von Werten für das Diagramm oder ähnliches.

20.3.2 Der Controller

Paket: `de.ls10.effects.core.plugin`

Der Controller, welcher in der `fragment.xml` spezifiziert wurde, bildet das Kernstück des Diagramms. Er wird durch den `PluginInitializer` erzeugt und sorgt unter anderem dafür, dass alle Zugriffe auf das Datenmodell gesteuert, sowie eigene Ressourcen an das *Eclipse* interne Datenmodell gebunden werden. Diese Bindung ist die Schnittstelle zum eigenen Datenmodell. Der Controller erbt von der Klasse `PluginController` aus dem Paket `plugin` und bekommt zur Identifizierung einen eindeutigen Namen zugewiesen. Dies geschieht durch Überschreiben der Methode `getQualifiedName`.

Des Weiteren muss die Methode `updateEditors` überschrieben werden, um den aktuellen Editor über Änderungen in der Szene zu informieren. Hierzu ist es nötig, die aktuelle Szene mittels `setScene` zu ändern.

Der oben erwähnte `PluginInitializer` muss in der Methode `getInitializerClass` des Controllers angegeben werden. Hierzu kann man den Befehl `Class.forName(...)` verwenden.

20.3.3 Der Plugin Initializer

Paket: `de.ls10.effects.core.plugin`

Die von `PluginInitializer` erbende Klasse initialisiert den Controller. Hierzu müssen verschiedene Methoden überschrieben werden:

- `registerMyClasses` lädt alle vom Diagramm benötigten Klassen via Reflection ein. Ein Codebeispiel befindet sich in der API-Dokumentation.
- `getXMLFileURL` gibt die XML-Datei an, welche für das Mapping zwischen Container und der `IResource` des JDT zuständig ist. Eine `IResource` in *Eclipse* ist eine Modellklasse, die entweder ein Project, ein Verzeichnis, eine Datei oder den Vaterknoten aller Projekte repräsentiert. Der Vaterknoten wird auch als *Workspace Root* bezeichnet.
- `init` erstellt eine Instanz des Plugin Controllers, wobei eine leere Szene erzeugt wird.
- `generateDiagram` arbeitet im Prinzip wie `init`, es wird jedoch die in einer vorherigen Sitzung gespeicherte Szene wiederhergestellt.
- `getClassName` muss den vollqualifizierten Namen der Klasse zurückliefern. Dies wird für die Serialisierung beim Speichern des Diagramms benötigt.
- `getInitializerDataResource` muss den Namen der Ressource zurückliefern, welcher in dem Workspace Root gesucht werden soll, während das Diagramm geladen wird.

20.3.4 Die Beschreibung der Szene

Paket: `de.ls10.effects.core.view3d`

Die Szenebeschreibung wird in einer Klasse realisiert, welche von `Scene` erbt. In dieser Klasse ist der `ContentBranch` des Szenegraphen aufzubauen, der dargestellt werden soll. Dazu ist die Methode `getContenBranch` zu implementieren, welche die `BranchGroup` mit der darzustellenden 3D-Szene zurückliefert. Diese Methode wird vom Editor aufgerufen, um die die Szene darstellende `BranchGroup` an das Universum zu binden und damit anzeigen zu können. Des Weiteren gibt es zu überschreibende Methoden, welche Parameter des Universums zurückliefern. Folgende Parameter stehen hierbei zur Verfügung:

- Höhe des Universums
- Breite des Universums

- Tiefe des Universums
- Startposition der Kamera im Universum

20.3.5 Das Datenmodell

Paket: `de.ls10.effects.core.datamodel`

Die abstrakte Klasse `Container` bietet die Basisfunktionen an, um das *Eclipse*-Datenmodell erweitern und diagrammspezifische Informationen in einem eigenen Datenmodell speichern zu können. Die erbende Klasse muss die Methode `getQualifiedName` überschreiben, die den Namen des entsprechenden Containers zurückliefert.

20.3.6 Anlegen eigener graphischer Objekte

Paket: `de.ls10.effects.core.view3d`

Sollen neben den im Basisplugin vorhandenen graphischen Primitiven eigene implementiert werden, so müssen diese von der abstrakten Klasse `GraphicalObjects` erben. Dies ist wichtig, da hierbei Eigenschaften der Transformgruppen gesetzt werden, die bei der Interaktion benötigt werden. Des Weiteren werden einige Standardfunktionen für graphische Objekte zur Verfügung gestellt.

20.4 Fazit

In diesem Text wurden nur die wesentlichen Punkte zur Implementierung eines eigenen Diagrammtyps aufgeführt. Je nach Anforderung kann es möglich sein, weitere Funktionalität zu implementieren. Hier bietet ***EFFECTS*** eine Sammlung von Basisklassen an, die zur freien Verfügung gestellt werden. Zum Beispiel existieren Klassen zur Realisierung eines Zustandsautomaten oder zum verschieben von Objekten. Nähere Informationen befinden sich in der API Dokumentation und der Beschreibung der Systemarchitektur.

TEIL 5

Fazit

KAPITEL 21

Reflexion über das Vorgehensmodell und die PG-Organisation

Semih Sevinç, Michael Nöthe

Im Folgenden wird über das in der Einleitung (s.Kapitel 2.7) erläuterte Vorgehensmodell diskutiert. Es werden die Vor -und Nachteile der benutzen XP-Techniken diskutiert und einige Verbesserungsvorschläge angegeben. Danach erfolgt eine Reflektion über die allgemeine Organisation der Projektgruppe.

21.1 Vorgehensmodell

Die einzelnen Tasks sollten von jeweils zwei Entwicklern mittels *Pair-Programming* bearbeitet werden, damit jeder an möglichst allen Aufgaben des Projektes beteiligt ist. Im ersten Release wurden nach jedem abgearbeiteten Task die Paare gewechselt. Aber bereits ab dem zweiten Release kam es nicht mehr zu immer komplett wechselnden Paaren. Es bildeten sich meist Teilgruppen in den einzelnen Aufgabengebieten. Die Teilgruppen setzen sich aus mindestens einem Experten und wechselnden Entwicklern zusammen. Gründe hierfür waren die Verteilung des Expertenwissens für das jeweilige Themengebiet und die Zeitersparnis, da der Experte bereits in dem Themengebiet eingearbeitet war und die übrigen Entwickler anleiten konnte. Dadurch wurde das Expertenwissen in begrenzten Maße weitergegeben. Ein großer Vorteil beim *Pair-Programming* war die Fehlersuche zu zweit und die gegenseitige Ergänzung bei der Einarbeitung in ein neues Thema.

Auch die Unterteilung des Projektes in mehrere Releases und die damit einhergehende fortlaufende Integration erwies sich als vorteilhaft, da man ständig eine ausführbare Version des Frameworks hatte, deren Funktionalität kontinuierlich weiterentwickelt wurde, und sich die Arbeitsbelastung der PG gleichmäßig auf das gesamte Jahr verteilte. Der Verzicht auf die umfassende und detaillierte Planung des Gesamtprojektes war unbedingt nötig, weil die benutzen Technologien, wie etwa *Eclipse* oder *Java3D*, zu komplex waren, um sie von vornherein in die Planung mit einzubeziehen. Außerdem hatte man durch den stetigen Wechsel zwischen Planungs- und Entwicklungsphasen stets abwechslungsreiche Aufgaben. Als Kritik wurde angemerkt, dass die PG-Teilnehmer mehr in die Gesamtplanung der Releases integriert werden sollten. Dadurch würden die Teilnehmer einen besseren Überblick über den Gesamtumfang und der Planung der Projektgruppe gewinnen. Darüber hinaus wurde die Reihenfolge der Releases (s. Kapitel 2.6) kritisiert. Während der Realisierung der Interaktion war eine komplette Überarbeitung des bereits existierenden Frameworks nötig, was zusätzliche Entwicklungszeit

kostete. So hätte man nach dem Erstellen der dreidimensionale Szene, zunächst auf die Interaktion eingehen und zuletzt das Framework realisieren sollen.

Die *Task Cards* boten eine gute Übersicht der zu erledigenden Aufgaben, wobei die Zeitabschätzung oft nicht realistisch waren, da der Umgang mit neuen Technologien unvorhersehbare Probleme mit sich brachte. Die Ableitung der *Task Cards* aus den *User Stories* war dabei nicht immer trivial, so dass sehr viele Tasks erst später aufgestellt wurden. Dies lässt darauf schließen, dass es sinnvoll sein könnte, einen Zwischenschritt einzuführen, in dem aus den *User Stories* zumindest eine grobe Architektur entworfen wird, ehe die Tasks identifiziert werden.

Es war zunächst geplant, das Projekt mit dem *Test-First* Ansatz zu bearbeiten, d.h. es wird zunächst ein Test geschrieben und dann erst die Klasse implementiert. Diese Technik erwies sich jedoch als nicht praktikabel, was mehrere Gründe hatte. Der *Test-First* Ansatz setzt eine detailierte Planung der zu implementierenden Klassen voraus. Für diese genaue Planung im Voraus fehlte neben der erforderlichen Zeit auch die Erfahrung der Entwickler, mit diesem Ansatz zu arbeiten. Jedoch wurden zu Beginn *JUnit*-Tests nach der Implementierung der Klasse geschrieben. Aber aufgrund der ständigen *Milestone*-Wechsel von *Eclipse*, und damit der verwendeten API, waren die *JUnit*-Tests ab dem dritten Release nicht mehr möglich.

Das Prinzip des *Kunden vor Ort* wurde zwar benutzt, aber nicht so wie es in XP vorgegeben ist. Kunden- und Entwicklerrolle sind oft zusammengefallen. Es wurde festgestellt, dass die Kunden eigentlich die Rolle eines Mediators zwischen Entwicklern und den PG-Betreuern haben, die die eigentlichen Kunden im XP-Sinne sind. Die Mediatoren verfügen als Entwickler über technisches Wissen und als Kunden über Domainwissen im Anwendungsbereich. Daher sind sie in der Lage bei auftretenden Schwierigkeiten in der Implemetierung zwischen den Anforderungen der PG-Betreuer und den Problemen der Entwickler zu vermitteln. Somit sind die Kunden eigentlich Entwickler in einer Sonderrolle.

21.2 Allgemeine Organisation der Projektgruppe

Nach den XP-Techniken wird nun die allgemeine Organisation der Projektgruppe diskutiert.

Die am Anfang der Projektgruppe durchgeführte Seminarfahrt war sehr sinnvoll zum Kennenlernen der anderen PG-Teilnehmer. Ein Verbesserungsvorschlag für die Seminarphase wäre, dass neben den Seminarvorträgen am ersten Tag eine XP-Vorstellung und eine *eXtreme Hour* nach (Beck, 2000; Beck und Fowler, 2001) am zweiten Tag statt finden soll. Außerdem wäre es angebrachter, bei den Vorträgen statt in die Themenbreite in die Thementiefe zu gehen, auch wenn dadurch nicht jeder PG-Teilnehmer einen Vortrag halten kann. Eine weitere Anregung wäre, dass man in der Woche nach dem Seminar gemeinsam eine Planung des Gesamtziels der PG als Workshop durchführt, um eine gemeinsame Systemmetapher zu entwickeln.

Im ersten Semester gab es neben der eigentlichen Implementierungsarbeit auch zwei Gruppen, die die Aufgabe hatten, Konzepte für die Syntax und Semantik der Modellierungsnotationen und die Benutzungsschnittstelle zu entwickeln. Jedoch wurden die Ergebnisse dieser beiden Gruppen nicht in der weiteren Arbeit der PG verwendet.

Die Erstellung der Dokumentation des PG-Berichts war stets releasebegleitend. Dies hatte

den sehr großen Vorteil, dass man nicht erst gegen Ende der Projektgruppe alles aufschreiben musste. Des Weiteren war dieses Vorgehen auch sinnvoll, da man nach jedem abgeschlossenen Release die einzelnen Details noch im Gedächnis hatte.

Die Gruppensitzungen fanden zweimal pro Woche statt. Nach dem zweiten Release wurde die Rolle des *Chief of the week* eingeführt. Dieser hatte die Aufgabe, in jeder Sitzung über den Stand der Dinge zu berichten. Diese Rolle war sehr vorteilhaft, sowohl für die Betreuer, als auch für die Entwickler, weil beide einen Ansprechpartner hatten, der einen Gesamtüberblick über den aktuellen Stand des Projektes besaß.

Die Betreuer sorgten durch ihre Aufgeschlossenheit gegenüber Vorschlägen der Teilnehmer und ihre konstruktive Kritik für ein sehr angenehmes Arbeitsklima. Auch die Teilnehmer harmonierten gut miteinander.

KAPITEL 22

Ausblick

Semih Sevinç, Michael Nöthe

In diesem Abschnitt wird zunächst der momentane Stand des Frameworks und des *ClassPackageDiagram*-Plugins erläutert. Des Weiteren wird auf die fehlenden Funktionalitäten und Erweiterungsmöglichkeiten des Frameworks und des *ClassPackageDiagram*-Plugins eingegangen. Am Ende werden anhand eines kleinen Beispielprojekts die Unterschiede und Vorteile zwischen zweidimensionaler (UML-) Notation und der dreidimensionalen Notation des ***EFFECTS***-Plugins dargestellt.

22.1 Momentaner Stand

Das *ClassPackageDiagram* stellt eine Integration von Klassen- und Paketdiagrammen dar. Es kann zum einem aus vorhanden Quellcode die Visualisierung der statischen Struktur eines Softwaresystems automatisiert erzeugt, zum anderen ein neues Projekt graphisch angelegt und bearbeitet werden. Aus dieser selbst erstellten Szene ist es dann möglich, den Quellcode zu erzeugen. Das Framework bietet eine Infrastruktur, die es ermöglicht, effizient neue dreidimensionle graphische Notationen zu implementieren.

22.2 Erweiterungsmöglichkeiten

Da es sich bei den einzelnen Diagramm-Plugins um prototypische Umsetzungen von Visualisierungen handelt, gibt es noch einige Schwachstellen bzw. Erweiterungsmöglichkeiten. Zunächst wird das *ClassPackageDiagram* betrachtet. Ein wichtiges Feature, das in diesem Diagramm fehlt, ist die Möglichkeit, Kollisionsabfragen zwischen Objekten im dreidimensionalen Raum durchzuführen. Dadurch wären folgende Funktionen möglich:

- Umstrukturierung auf graphischer Ebene: Die Zugehörigkeit eines Elementes zu einem Paket soll anhand ihrer räumlichen Lage in der Szene bestimmt werden können, d.h. es soll z.B. möglich sein, einen Klassenwürfel in ein Paketwürfel zuziehen und dadurch auf Codeebene eine Javaklasse in ein Paket zu verschieben. Weiterhin sollten Cone Trees graphisch editierbar sein, insbesondere soll man eine Klasse in einen Cone Tree „einhängen“ können, in dem man sie an die Unterseite eines Cones verschiebt. Um die räumliche Nähe von Klassenwürfeln und Cone Trees, sowie die Lage eines Klassenwürfels in einem Paketwürfel erkennen zu können, ist eine Kollisionsabfrage erforderlich.

- Syntaxprüfung auf graphischer Ebene: Es ist im *ClassPackageDiagram* möglich, Elemente so anzuordnen, dass aus dem Diagramm kein korrekter Javacode erzeugt werden kann. Solche Fälle treten z.B. auf, wenn man zwei Klassen ineinander verschiebt, oder eine Klasse so anordnet, dass sie teilweise innerhalb und teilweise außerhalb eines Paketwürfels liegt. Eine Überprüfung der Kollision zwischen den zu verschiebenen Würfeln kann diese Fälle in der Szene erkennen und dem Benutzer melden.

Ein weiteres nicht implementiertes Feature betrifft Beziehungen über Paketgrenzen hinweg. Im Falle von Vererbungsbeziehungen war es vorgesehen, sogenannte „Proxies“ zu benutzen (siehe 17.2), für alle anderen Arten von Beziehungen sollten „paketübergreifende Assoziationen“ (siehe 17.2) verwendet werden.

Auch das den Diagrammen zugrundeliegende Framework ist noch verbesserungsfähig. Zu nennen wären hier die Navigation, die man etwas intuitiver gestalten könnte. Es kann möglicherweise vorteilhaft sein, ein dreidimensionales Eingabegerät zu verwenden. Eine Studie zur Bewertung der Benutzerführung könnte wertvolle Hinweise zur Verbesserung der Bedienbarkeit des ***EFFECTS***-Tools liefern. Dies war jedoch im Rahmen der Projektgruppe nicht möglich. Die bereits erwähnte Kollisionsabfrage beim *ClassPackageDiagram* sollte in das Framework mit aufgenommen werden, damit dieses Feature ebenfalls für weitere Diagramm-Plugins zur Verfügung steht.

22.3 Beispiel

Zum Schluss wird anhand eines kleinen Beispiels versucht darzustellen, dass die dreidimensionale Visualierung von Klassenstrukturen im Gegensatz zur zweidimensionlen Darstellung vorteilhafter ist. Dabei wird ein kleines Beispielprojekt erzeugt, wobei schon zugunsten einer übersichtlicheren Darstellung in der zweidimensionalen UML-Notation auf eine komplexe Paketverschachtelung verzichtet wurde. Dieses Beispielprojekt wurde zunächst in 2D (s. Abbildung 22.1) mit dem Modellierungswerkzeug *Omondo* und dann in 3D (s. Abbildung 22.2) mit ***EFFECTS*** visualisiert.

Sicherlich ist dieses kleine Beispiel nicht sehr aussagekräftig und repräsentativ für alle Anwendungsfälle. Dazu müsste man eine empirische Studie mit zahlreichen Beispielen durchführen, indem man auch andere Diagrammarten mit einbezieht. Aber es wird in diesem Beispiel doch sehr deutlich, dass die Anordnung der Klassen in einem Cone Tree die Übersichtlichkeit sehr stark verbessert. Vor allem die Erbungshierarchie wird sehr deutlich und überschaubarer als die Darstellung in 2D. Auch die Möglichkeit, dass man in der dreidimensionalen Darstellung zusätzlich die Pakete visualieren kann, zu denen die Klassen gehören, trägt sicherlich einiges zum besseren Codeverständnis bei.

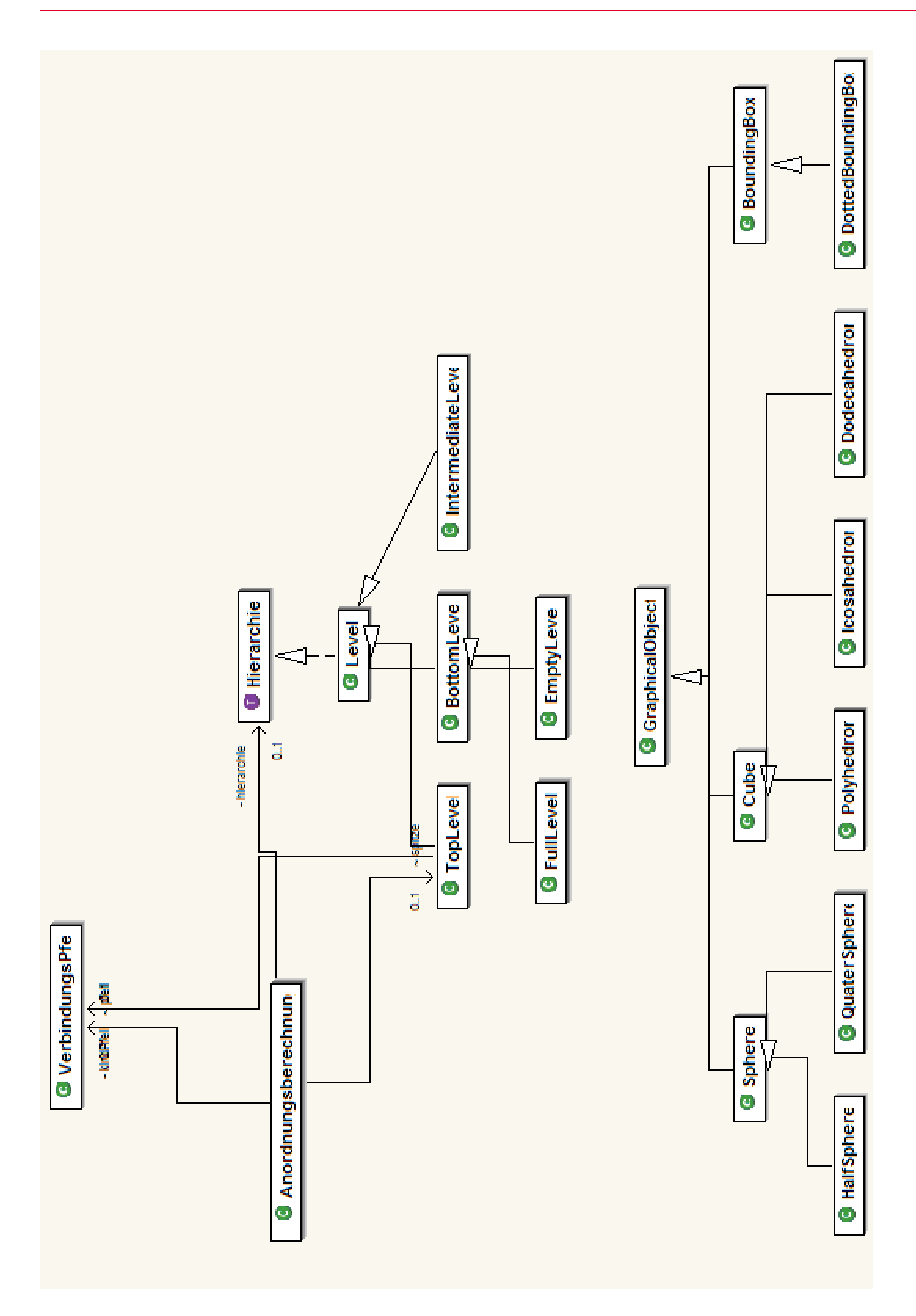


Abbildung 22.1.: Zweidimensionale Darstellung von Klassenstrukturen mit Omondo

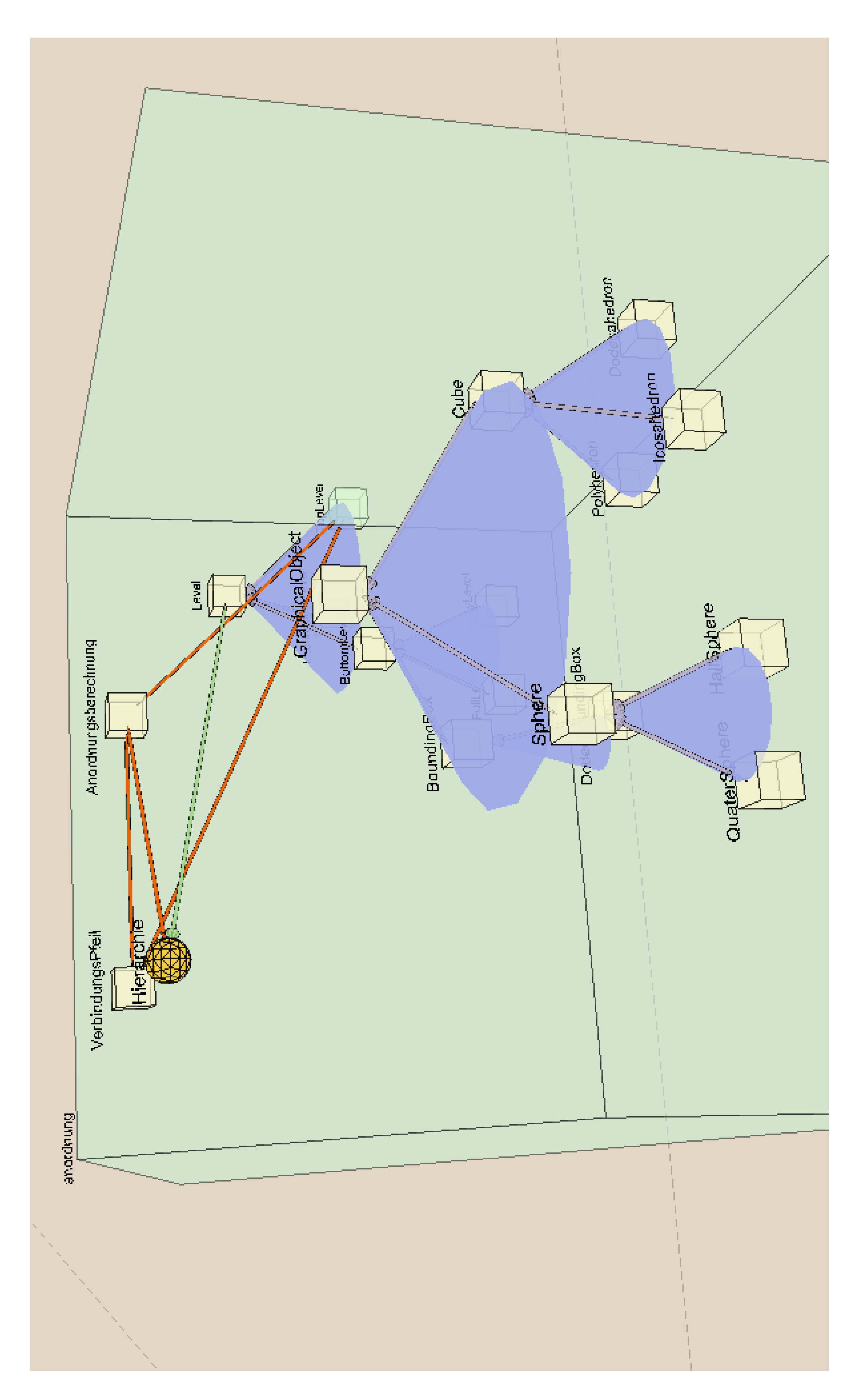


Abbildung 22.2.: Dreidimensionale Darstellung von Klassenstrukturen mit ***EFFECTS***

TEIL 6

Anhang

ANHANG A

Code Konventionen

Christian Mocek

A.1 Namenskonventionen

A.1.1 Pakete

Paketnamen werden grundlegend klein geschrieben.

A.1.2 Klassen und Interfaces

1. Namen beginnen mit einem Großbuchstaben und jedes interne Wort ebenfalls.
2. Abkürzungen müssen vermieden werden, soweit diese nicht bekannt sind.
3. Der Name sollte kurz aber präzise und einprägsam sein.

A.1.3 Methoden

Die Konventionen sind die der Klassen und Interfaces, wobei jedoch der erst Buchstabe klein geschrieben wird.

A.1.4 Variablen

1. Die Konventionen sind die der Klassen und Interfaces, wobei jedoch der erst Buchstabe klein geschrieben wird.
2. Die Namen dürfen nicht mit _ oder $ anfangen.

A.1.5 Konstanten

1. Alle Buchstaben werden groß geschrieben.
2. Einzelne Wörter werden durch _ getrennt.

A.2 Aufbau der Java Dateien

1. Die zu verwendende Sprache ist Englisch.
2. Jeder der Vorkommenden und hier besprochenen vier Abschnitte einer Java-Datei wird durch *zwei* Leerzeilen getrennt.
3. Eine Datei sollte auf keinen Fall mehr als 2000 Zeilen beinhalten.
4. Für jede Klasse bzw. für jedes Interface wird eine Java-Datei angelegt.

Jede Datei besteht aus den folgenden Abschnitten in der hier angegebenen Reihenfolge:

1. Anfangskommentar
2. Paketdefinition
3. *import*-Anweisungen
4. Klassen- und Interfacedeklarationen

A.2.1 Der Anfangskommentar

Jede Datei enthält einen Anfangskommentar als ersten Abschnitt, welcher wie folgt aussieht:

```
/*
 * Classname    :
 *
 * Version      :
 *
 * Create date  :
 * Last changed :
 *
 * Copyrightinformationen
 */
```

A.2.2 Die Paketdefinition

Der zweite Abschnitt enthält gegebenenfalls eine Paketdefinition.

```
package paketname;
```

A.2.3 Die Importanweisungen

Alle Paketimporte werden im dritten Abschnitt angegeben. Hierbei ist zu beachten, dass Importanweisungen aus einem Paket in der Reihenfolge zusammenstehen.

```
// FALSCH
import java.awt.color;
import java.lang.ref;
import java.awt.geom;

// RICHTIG
import java.awt.color;
import java.awt.geom;
import java.lang.ref;
```

A.2.4 Klassen- und Interfacedeklarationen

Im folgenden Abschnitt werden die Teile einer Klassen- bzw. Interfacedeklaration in der Reihenfolge ihres Auftretens erläutert:

1. Dokumentation der Klasse bzw. des Interface
2. Deklaration
3. Non-Javadoc Implementierungsdokumentation (optional)
4. statische Klassenvariablen
5. Instanzvariablen
6. Konstruktoren
7. Methoden

Dokumentation der Klasse bzw. des Interface

1. Die Dokumentation folgt den Javadoc-Richtlinien und wird mit `/** ... */` eingeleitet.
2. Die Dokumentation sollte knapp, aber präzise die Funktion der Klasse wiedergeben.
3. Gegebenenfalls sollte ein kurzer Beispielcode die Anwendung der Klasse verdeutlichen.

Deklaration

Nach der Dokumentation folgt die Deklaration der Klasse bzw. des Interface.

Non-Javadoc Implementierungsdokumentation

Gegebenenfalls folgt nach der Deklaration eine Dokumentation in der Informationen stehen, die nicht in die Javadoc-Dokumentation enthalten sein sollen. Eingeleitet wird diese mittels `/*... */`.

statische Klassenvariablen

Die Reihenfolge der *static* Klassenvariablen ist:

1. *public*
2. *protected*
3. Paketvariablen. Diese besitzen keinen Zugriffsbezeichner.
4. *private*

Instanzvariablen

Die Reihenfolge der Instanzvariablen ist genau wie bei den statischen Klassenvariablen.

Konstruktoren

Bei überladenen Konstruktoren sollten diese in aufsteigender Reihenfolge der möglichen Übergabeparameter angegeben werden.

```
public class Klasse extends Irgendwas {
    ...
    Klasse();
    Klasse(String a);
    Klasse(String a, int b);
    ...
}
```

Methoden

Um das Verständnis der Klasse zu erleichtern, werden die Methoden nach Funktionalität gruppiert.

A.3 Leerzeilen und Leerzeichen

A.3.1 Leerzeilen

Zwei Leerzeilen werden eingefügt:

1. Zwischen den Hauptabschnitten der Java-Datei

Eine Leerzeile wiederum wird in den folgenden Fällen eingefügt:

1. Vor jedem Kommentar ist eine Leerzeile einzufügen.
2. Folgt nach einer Deklaration einer Variablen keine weitere, so wird eine Leerzeile eingefügt.
3. Methodendeklarationen werden durch eine Leerzeile getrennt.

4. Zwischen jeder *case*-Marke ist eine Leerzeile, außer es folgen mehrere case-Marken direkt aufeinander.
5. Zwischen logischen Abschnitten in einer Methode.

A.3.2 Leerzeichen

1. Ein Leerzeichen hinter einem Komma
2. Ein Leerzeichen hinter einem Semikolon in einer for-Schleife.
3. Binären Operatoren werden von ihrem Operanden durch ein Leerzeichen getrennt.
4. Unäre Operatoren werden *nicht* von ihrem Operanden getrennt.

```
a += c + d;
a = (a + b) / (c + d);

while(d++ = s++) {
    n++;
}
prints("size␣is" + foo + "\n");
```

A.4 Die Einrückung

1. Die Tabweite beträgt exakt 4 Leerzeichen, wobei Tabs den Leerzeichen vorzuziehen sind.
2. Beispielcode in der Klassendokumentation sollte maximal 70 Zeichen lang sein.
3. Eine Zeile darf nicht mehr als 255 Zeichen lang sein.

A.4.1 Codeblöcke

Codeblöcke werden durch { und } gekennzeichnet. Diese öffnende Klammer muss auf gleicher Ebene eingerückt sein wie die geschlossene Klammer, es sei denn es handelt sich um einen leeren Block.

```
// RICHTIG
if(irgendwas == true) {
    ...
}

// FALSCH
if(irgendwas == true)
{
    ...
}
```

A.4.2 Zeilenumbrüche

Falls eine Codezeile umgebrochen werden soll, so gelten folgende Regeln für den Zeilenumbruch:

1. Nach einem Komma
2. Vor einem Operator
3. High-Level sind Low-Level Zeilenumbrüche vorzuziehen.
4. Die nächste Zeile wird
   a) bei Zuweisungen auf das Niveau des Zuweisungsoperators plus ein Leerzeichen eingerückt
   b) bei Funktionsaufrufen auf das `(`-Zeichen plus ein Leerzeichen eingerückt

A.4.3 Beispiele

Methodenaufrufe

```
methodeEins(int a, int b, int c, String buffer,
            String output, int x);

x = methodeZwei(String a, String b,
                methodeDrei(int y));
```

Zuweisungsoperatoren

Handelt es sich um Zuweisungsoperatoren, gilt das folgende Beispiel. Hierbei kann man auch den Unterschied zwischen High-Level und Low-Level erkennen.

```
// FALSCH
wertEins = wert2 * (wert3 + wert4 -
                    wert5) + 4 * wert6 - wert7;
// RICHTIG
wertEins = wert2 * (wert3 + wert4 - wert5)
           + 4 * wert6 - wert7;
```

Methodendeklarationen

```
methodeX(int a, Object obj, String buffer,
         Object obj2) {
...
}

private static synchronized methodeY(int a,
                                     Object obj1,
                                     String buffer,
```

```
                                        Object obj2) {
...
}
```

Der ternäre Operator

```
alpha = wahr ? beta : gamma;

alpha = wahr ? beta
             : gamma;

alpha = wahr
        ? beta
        : gamma;
```

If-Statements

```
// FALSCH
if((Bedingung1 && Bedingung2) ||
   || (Bedingung3 && Bedingung4)
   !(Bedingung5 && Bedingung6)   ||
   (Bedingung7 && Bedingung7)) {
    macheIrgendetwas();
}

// RICHTIG
if((Bedingung1 && Bedingung2)
   || (Bedingung3 && Bedingung4)
   ||!(Bedingung5 && Bedingung6)) {
    macheIrgendetwas();
}

if((Bedingung1 && Bedingung2) || (Bedingung3 && Bedingung4)
   ||!(Bedingung5 && Bedingung6)) {
    macheIrgendetwas();
}
```

A.5 Dokumentation

Es existieren zwei Arten von möglichen Kommentarblöcken: *Implemetierungskommentare*, sowie *Dokumentationskommentare* im Javadoc-Format.

1. Ziel der Implementierungskommentare ist es, dem Entwickler ein besseres Verständnis des Source-Codes zu geben. Diese Informationen sind nicht in einer externen Dokumentation zu finden.

2. Dokumentationskommentare im Javadoc-Format enthalten nur solche Informationen, die nichts mit der konkreten Implementierung der Klasse zu tun haben, sondern nur die

Funktionalität der Klasse und deren Methoden beschreiben.

Alle Kommentare müssen den hier angegeben Regeln folgen:

1. Kommentare enthalten nur solche zusätzlichen Informationen, die nicht trivial aus dem Source-Code entnommen werden können.

2. Kommentare enthalten nur Informationen, die absolut notwendig zum Verständnis des Programms sind.

3. Alle öffentlichen, also als *public* deklarierten Elemente müssen mit einem Dokumentationskommentar ausgestattet sein.

4. Informationen über das Design sind in sofern angebracht, als dass diese nicht-trivial bzw. nicht-eindeutig aus dem Source-Code entommen werden können.

   Achtung: Aufgrund des extreme-Programming Gedankens sollten solche Informationen möglichst nicht zu umfangreich und speziell eingebunden werden, da diese Informationen zu schnell veralten können!

Allgemein gilt: Vor jedem Kommentar ist eine Leerzeile einzufügen.

A.5.1 Aufbau der Implementierungskommentare

Programme können zwei verschiedene Arten von Implementierungskommentaren enthalten.

mehrzeilige Blockkommentare

1. Werden mittels `/*...*/` eingeleitet.

2. Das erste Zeichen jeder Zeile eines Blockkommentars ist ein Asterix (`*`).

3. Jeder Blockkommentar hat mindestens drei Zeilen.

4. Der Kommentar ist auf die gleiche Tiefe eingerückt wie der Code.

```
/*
 * Dies ist ein Blockommentar.
 */
```

Nicht erlaubt ist es, einen Blockkommentar in eine Zeile zu schreiben:

```
/* Dies ist ein kein Blockommentar!!! */
```

einzeilige Kommentare

1. Werden mittels `//` eingeleitet.

2. Es dürfen keine mehrzeiligen Kommentare in dieser Form geschrieben werden.

3. Der Kommentar ist auf die gleiche Tiefe eingerückt wie der Code.

4. Dürfen nicht ans Ende einer Codezeile gestellt werden, es sei denn, es handelt sich um die Deklaration einer *private* oder *protected* Klassen- bzw. Instanzvariablen.

5. Der `//`-Begrenzer darf genutzt werden um mehrere Zeilen Code auszukommentieren. Hierbei muss das `//` am Anfang der Zeile stehen.

```
if(wahr) {
    // hier wird was deklariert
    int alpha = 10;

    // Das hier brauchen wir nicht mehr
//  for(int i = 0; i < alpha; i++)
//  {
//      System.out.println(i.toString());
//  }
}
else {
    return false;
}
```

A.5.2 Aufbau der Dokumentationskommentare

Dokumentationskommentare folgen den Regeln der Javadoc-Spezifikation. Diese ist zu finden unter

http://java.sun.com/products/jdk/javadoc/writingdoccomments.html

Für die öffentlichen Dokumentationskommentare gelten die folgenden Regeln:

1. Werden mittels `/**...*/` eingeleitet.

2. Das erste Zeichen jeder Zeile eines Kommentars ist ein Asterix (`*`).

3. Jeder Kommentar hat mindestens drei Zeilen.

4. Der Kommentar ist auf die gleiche Tiefe eingerückt wie der Code.

5. Steht direkt vor der Deklaration einer Methode, einer Klasse, eines Interface und eines Feldes, wenn dies in die öffentliche Dokumentation mit aufgenommen werden soll und niemals innerhalb einer Deklaration.

```
/**
 * Diese Klasse tut nichts...
 */
public class IrgendeineKlasse {
    /**
     * Den braucht man eigentlich nicht.
     */
    public String buffer;
```

```
    ...

    /**
     * Die Methode ist sinnlos
     * @param wert ein sinnloser Eingabeparameter
     * @return der eingegebene Wert.
     */
     int getEingabe(int wert) {
        return wert;
     }
}
```

A.6 Deklarationen

A.6.1 Deklarationen pro Zeile

1. Pro Zeile wird maximal eine Variable deklariert.
2. Bei einem Block von nicht-öffentlichen Variablen, bei denen der Kommentar an das Ende der Zeile verschoben wurde, müssen alle Variablennamen und alle Kommentare untereinander angeordnet sein.
3. Folgt nach einer Deklaration keine weitere, so wird eine Leerzeile eingefügt.
4. Variablen mit gleichem Namen sollten vermieden werden, um die Variable auf höherem logischem Level nicht zu überdecken.
5. Die Sichtbarkeit sollte für Methoden und Variablen möglichst minimal sein. Es ist also wo immer möglich *private* zu verwenden.

```
int     level;          // Ebene des Eintrags
int     size;           // Groesse des Eintrags
Object  currentEntry;   // Der selektierte Eintrag
```

A.6.2 Anordnung der Deklarationen

Deklarationen werden immer zu Beginn eines Codeblocks eingefügt. Im Code deklarierte Variablen sind nicht zulässig.

Ausnahme: Bei einer *for*-Schleife darf die Zählvariable lokal deklariert werden.

```
void methodeEins(int eingabe) {
    int index = 10;

    if(index < eingabe) {
        String buffer("temp");
```

```
        ...

        for(int i = 0; i < eingabe; i++);
    }
}
```

A.6.3 Initialisierungen

Variablen die nicht von berechneten Werten abhängen, müssen direkt bei ihrer Deklaration initialisiert werden.

A.6.4 Klassen und Interfacedeklarationen

1. Zwischen den Methodennamen und seiner Parameterliste steht kein Leerzeichen.
2. Die Codeblöcke sind wie oben beschrieben zu behandeln.
3. Methodendeklarationen werden durch eine Leerzeile getrennt.

```
class Beispiel extends Object {
    int a;
    int b;

    Beispiel(int i, int j) {
        a = i;
        b = j;
    }

    void methodeEins() {}
}
```

A.7 Statements

1. Falls Statements teil einer Kontrollstruktur sind wie z.B. if-else oder for Statments, dann wird in jedem Fall ein Codeblock eingeführt.
2. Zwischen dem Schlüsselwort und der öffnenden Klammer steht kein Leerzeichen.

```
// FALSCH
if(wennwahrdann)
    statement;

// RICHTIG
if(wennwahrdann) {
    statement;
}
```

A.7.1 Einfache Statements

Jede Zeile enthält maximal ein Statement.

```
// FALSCH
arv++; argv--;

// RICHTIG
argv++;
argc++;
```

A.7.2 for und while Statements

Diese Strukturen können abgekürzt werden, wenn die Schleife leer. Hierzu werden die Codeblöcke durch ein ; ersetzt.

```
for(Initialisierung; Bedingung; Update);


while(Bedingung);
```

A.7.3 if, if-else, if-else-if-else Statements

Diese Kontrollstruktur hat die hier dargestellte Form.

```
if(wahr) {
    statement;
}

if(wahr) {
    statement;
}
else {
    statement;
}

if(wahr) {
    statement;
}
else if (bedingung) {
    statement;
}
else {
    statement;
}
```

A.7.4 try-catch Blöcke

Ein try-catch-Block hat die Form.

```
try {
    statement;
}
catch(ExceptionClass e) {
    statement;
}
```

A.7.5 switch Statements

1. Jedes *switch*-Statement besitzt eine *default*-Marke.
2. Jede *default*-Marke besitzt ein *break*-Kommando.
3. die *default*-Marke ist immer die letzte Marke im switch-Statement.
4. Falls in einem *case* kein *break* verwendet wird, wird an dessen Stelle ein Kommentar hinzugefügt der darauf hinweist. Dieser Kommentar muss auf höhr der *case*-Anweisung eingerückt sein.
5. Zwischen jeder *case*-Marke ist eine Leerzeile.

Ausnahme: Falls mehrere *case*-Marken aufeinander folgen, gelten die Punkte 4 und 5 nicht.

```
switch(Bedingung) {
    case ABC:
        statement;
    // Kein break!

    case DEF:
        statement;
        break;

    case GHI:
    case JKL:
        statement;
        break;

    default:
        statement;
        break;
}
```

The GNU General Public License

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect

making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

Terms and Conditions For Copying, Distribution and Modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
   b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
   c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the

Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

No Warranty

11. Because the program is licensed free of charge, there is no warranty for the program, to the extent permitted by applicable law. Except when otherwise stated in writing the copyright holders and/or other parties provide the program "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair or correction.

12. In no event unless required by applicable law or agreed to in writing will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

End of Terms and Conditions

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

> one line to give the program’s name and a brief idea of what it does.
> Copyright (C) yyyy name of author
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) yyyy name of author
> Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
> This is free software, and you are welcome to redistribute it under certain conditions; type ‘show c’ for details.

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w` and `show c`; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program
> ‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Literaturverzeichnis

[Alexander u. a. 1977] ALEXANDER, Christopher ; ISHIKAWA, Sara ; SILVERSTEIN, Murray: *A Pattern Language*. Oxford University Press, 1977

[Barrilleaux 2001] BARRILLEAUX, Jon: *3D user interfaces with Java 3D*. Manning, 2001

[Battista u. a. 1994] BATTISTA, Giuseppe D. ; EADES, Peter ; TAMASSIA, Roberto ; TOLLIS, Ioannis G.: Algorithms for Drawing Graphs: an Annotated Bibliography / Department of Computer Science, Brown University, USA. 1994. – Forschungsbericht

[Beck 1999] BECK, Kent: *Extreme Programming: A gentle introduction*. 1999. – URL http://www.extremeprogramming.org. – Zuletzt gesichtet: 19.08.2004

[Beck 2000] BECK, Kent: *eXtreme Programming*. Addison-Wesley-Verlag, 2000

[Beck und Fowler 2001] BECK, Kent ; FOWLER, Martin: *eXtreme Programming planen*. Addison-Wesley-Verlag, 2001

[Beck und Gamma 2001] BECK, Kent ; GAMMA, Erich: *JUnit*. 2001. – URL http://www.junit.org. – Zuletzt gesichtet: 19.08.2004

[Boles 1994] BOLES, Dietrich: *Das IMRA-Modell*, Fachbereich Informatik der Universität Oldenburg, Diplomarbeit, 1994

[Bolour 2003] BOLOUR, Azad: *Notes on the Eclipse Plug-In Architecture*. Juli 2003. – URL http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html. – Zuletzt gesichtet: 19.08.2004

[Booch u. a. 1999] BOOCH, Grady ; RUMBAUGH, Jim ; JACOBSON, Ivar: *Das UML-Benutzerhandbuch*. Addison-Wesley, 1999

[Brüderlein und Meier 2000] BRÜDERLEIN, Beat ; MEIER, Andreas: *Computergrafik und geometrisches Modellieren*. Teubner, 2000

[Buschmann u. a. 1996] BUSCHMANN, Frank ; MEUNIER, Regine ; ROHNERT, Hans ; SOMMERLAD, Peter ; STAL, Michael: *A System of Patterns*. John Wiley and Sons Ltd., 1996

[Chok und Marriott 1995] CHOK, Sitt S. ; MARRIOTT, Kim: Automatic construction of user interfaces from constraint multiset grammars. In: *Proceedings 11th IEEE symposium on Visual Languages – VL'95*, 1995, S. 242–249

[Computer Associates International, Inc. Islandia, USA 2000] COMPUTER ASSOCIATES INTERNATIONAL, INC. ISLANDIA, USA: *Cosmo Player 2.1*. erhältlich online. 2000. – URL http://www.cai.com/cosmo. – Zuletzt gesichtet: 19.08.2004

[Daum 2003] DAUM, Berthold: *Java–Entwicklung mit Eclipse 2*. dpunkt Verlag, Juni 2003

[Eclipse Foundation 2003] ECLIPSE FOUNDATION: *Eclipse Project Slide Presentation*. 2003. – URL http://eclipse.org/eclipse/presentation/eclipse-slides.html. – Zuletzt gesichtet: 19.08.2004

[Eclipse Foundation 2004] ECLIPSE FOUNDATION: *The Eclipse Project*. 2004. – URL http://www.eclipse.org. – Zugriffsdatum: 19.08.2004

[Engelen 2000] ENGELEN, Frank: *Konzeption und Implementierung eines dreidimensionalen Klassenbrowsers für Java*, Universität Dortmund, Fachbereich Informatik, LS10, Diplomarbeit, 2000

[Esser und Janneck 2001] ESSER, Robert ; JANNECK, Jorn W.: A predicate-based approach to defining visual language syntax. In: *Symposium on Visual Languages and Formal Methods*, 2001

[Furnas 1981] FURNAS, George W.: The Fisheye View: A New Look at Structured Files / Bell Laboratories. 1981. – Forschungsbericht

[Gamma u. a. 2001] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLIESSIDES, John: *Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, 2001

[Ghezzi u. a. 1999] GHEZZI, Carlo ; JAZAYERI, Mhedi ; MANDRIOLI, Dino: *Fundamentals of Software Engineering*. Prentice Hall, 1999

[Gil und Kent 1998] GIL, Y. ; KENT, S.: Three Dimensional Software Modelling. In: *Proceedings of ICSE98*, IEEE Press, September 1998. – URL http://www.cs.ukc.ac.uk/pubs/1998/790. – Zuletzt gesichtet: 19.08.2004

[Gille 1999] GILLE, Marc: *Diagramm-Editoren*. Spektrum, 1999

[Golin 1991] GOLIN, Eric J.: Parsing visual languages with picture layout grammars. In: *Journal of Visual Languages and Computing* 2 (1991), Nr. 4, S. 371–394

[Helm und Marriott 1991] HELM, Richard ; MARRIOTT, Kim: A Declarative Specification and Semantics for Visual Languages. In: *Journal of Visual Languages and Computing, Ausgabe 2* (1991), S. 311–331

[Helm u. a. 1991] HELM, Richard ; MARRIOTT, Kim ; ODERSKY, Martin: Building visual languages parsers. In: *Conference Proceedings on Human Factors in Computing Systems (CHI'91)* (1991), S. 105–112

[Herrmann 2001] HERRMANN, Thomas: *Kompendium zur Grundvorlesung Informatik und Gesellschaft*. Universität Dortmund, Fachbereich Informatik, Lehrstuhl für Informatik und Gesellschaft. 2001

[Hitz und Kappel 2002] HITZ, Martin ; KAPPEL, Gerti: *UML@work*. 2. Auflage. dpunkt–Verlag, 2002

[Kehn 2002] KEHN, Dan: *How to test your Internationalized Eclipse Plug-In*. August 2002. – URL http://www.eclipse.org/articles/Article-TVT/how2TestI18n.html. – Zuletzt gesichtet: 19.08.2004

[Kühne 2002] KÜHNE, Thomas: *Skript Programmiermethodik*. TU Darmstadt. 2002

[Leisering 1999] LEISERING, Horst: *Neues grosses Wörterbuch - Fremdwörterbuch*. Compact, München, 1999

[Link 2002] LINK, Johannes: *Unit Tests mit Java*. dpunkt-Verlag, 2002

[Lippert u. a. 2002] LIPPERT, Martin ; ROOCK, Stefan ; WOLF, Henning: *Software entwickeln mit XP*. dpunkt-Verlag, 2002

[Marriott 1994] MARRIOTT, Kim: Constraint multiset grammars. In: *IEEE Symposium on Visual Languages*, 1994

[Rechenberg 1999] RECHENBERG, Peter: Formale Sprachen und Automaten. In: RECHENBERG, Peter (Hrsg.) ; POMBERGER, G. (Hrsg.): *Informatik-Handbuch*. Carl Hanser Verlag München, 1999, S. 89–110

[Rechenberg und Pomberger 2002] RECHENBERG, Peter (Hrsg.) ; POMBERGER, Gustav (Hrsg.): *Informatikhandbuch*. Hanser, 2002

[Reichenberger und Steinmetz 1999] REICHENBERGER, Klaus ; STEINMETZ, Ralf: Visualisierungen und ihre Rolle in Multimedia-Anwendungen. In: *Informatik Spektrum* 22 (1999), S. 88–98

[Rekers und Schürr 1997] REKERS, Jan ; SCHÜRR, Andy: Defining and Parsing Visual Languages with Layered Graph Grammars. In: *Journal of Visual Languages and Computing* 8 (1997), Nr. 1, S. 27–55

[Rekimoto und Green 1993] REKIMOTO, Jun ; GREEN, Mark: The Information Cube: Using Transparency in 3D Information Visualization / Dep. of Computing Science, University of Alberta. 1993. – Forschungsbericht

[Reps und Teitelbaum 1989] REPS, Thomas W. ; TEITELBAUM, Tim: *The Synthesizer Generator*. Springer, 1989

[Robertson u. a. 1991] ROBERTSON, George G. ; MACKINLAY, Jock D. ; CARD, Stuart K.: Cone Trees: Animated 3D Visualizations of Hierarchical Information / Xerox PARC. 1991. – Forschungsbericht

[Royce 1970] ROYCE, Winston W.: *Managing the development of large software-systems*. Proc. IEE Wescon, 1970

[Schiedermeier 2002] SCHIEDERMEIER, Prof. Dr. R.: Die homogene Komponente bei Transformationen / Fachhochschule München. URL http://www.informatik.fh-muenchen.de/~schieder/graphik-01-02/slide0099.html, 2002. – Forschungsbericht. Zuletzt gesichtet: 19.08.2004

[Schiffer und Violka 2003] SCHIFFER, Bernd ; VIOLKA, Karsten: Spielzimmer aufräumen - Refaktorisieren macht Quellcode lesbarer. In: *c't Heft 17* (2003)

[Schürr und Westfechel 1992] SCHÜRR, Andy ; WESTFECHEL, Bernhard: Graphgrammatiken / RWTH Aachen – Fakultät für Informatik. 1992 (92–15). – Forschungsbericht

[Szwillus 1990] SZWILLUS, Gerd: *Specification of graphical structure editors*. Forschungsberichte des Fachbereichs Informatik der Universität Dortmund, 1990

[The Hillside Group] THE HILLSIDE GROUP: *Patterns Home Page: Your pattern library*. – URL http://hillside.net/patterns. – Zuletzt gesichtet: 19.08.2004

[UseNet-Gruppe 2004] USENET-GRUPPE: *Frequently Asked Questions (FAQ)*. UseNet. 2004. – URL news://comp.graphics.visualization

[Webnox Corporation 2003] WEBNOX CORPORATION: *Hyper Dictionary*. 2003. – URL http://www.hyperdictionary.com/computer. – Zuletzt gesichtet: 19.08.2004

[Westphal 2000] WESTPHAL, Frank: *Homepage*. 2000. – URL http://www.frankwestphal.de. – Zuletzt gesichtet: 19.08.2004

[Yacoub und Ammar 1998] YACOUB, Sherif M. ; AMMAR, Hany H.: Finite state machine patterns / Computer Science and Electrical Engineering Department, West Virginia University, Morgantown, West Virginia, WV26506. 1998. – Technical report

/125/ Ernst-Erich Doberkat
A Remark on A. Edalat's Paper *Semi-Pullbacks and Bisimulations in Categories of Markov-Processes*
Juli 2002

/126/ Alexander Fronk
Towards the algebraic analysis of hyperlink structures
August 2002

/127/ Markus Alvermann, Martin Ernst, Tamara Flatt, Urs Helmig, Thorsten Langer
Ingo Röpling, Clemens Schäfer, Nikolai Schreier, Olga Shtern
Ursula Wellen, Dirk Peters, Volker Gruhn
Project Group Chairware Final Report
August 2002

/128/ Timo Albert, Zahir Amiri, Dino Hasanbegovic, Narcisse Kemogne Kamdem,
Christian Kotthoff, Dennis Müller, Matthias Niggemeier, Andre Pavlenko, Stefan Pinschke,
Alireza Salemi, Bastian Schlich, Alexander Schmitz
Volker Gruhn, Lothar Schöpe, Ursula Wellen
Zwischenbericht der Projektgruppe Com42Bill (PG 411)
September 2002

/129/ Alexander Fronk
An Approach to Algebraic Semantics of Object-Oriented Languages
Oktober 2002

/130/ Ernst-Erich Doberkat
Semi-Pullbacks and Bisimulations in Categories of Stochastic Relations
November 2002

/131/ Yalda Ariana, Oliver Effner, Marcel Gleis, Martin Krzysiak,
Jens Lauert, Thomas Louis, Carsten Röttgers, Kai Schwaighofer,
Martin Testrot, Uwe Ulrich, Xingguang Yuan
Prof. Dr. Volker Gruhn, Sami Beydeda
Endbericht der PG nightshift:
Dokumentation der verteilten Geschäftsprozesse im FBI und Umsetzung von Teilen dieser Prozesse im Rahmen eines FBI-Intranets basierend auf WAP- und Java-Technologie
Februar 2003

/132/ Ernst-Erich Doberkat, Eugenio G. Omodeo
ER Modelling from First Relational Principles
Februar 2003

/133/ Klaus Alfert, Ernst-Erich Doberkat, Gregor Engels (Hrsg.)
Ergebnisbericht des Jahres 2002 des Projektes "MuSofT – Multimedia in der SoftwareTechnik"
März 2003

/134/ Ernst-Erich Doberkat
Tracing Relations Probabilistically
März 2003

/135/ Timo Albert, Zahir Amiri, Dino Hasanbegovic, Narcisse Kemogne Kamdem,
Christian Kotthoff, Dennis Müller, Matthias Niggemeier,
Andre Pavlenko, Alireza Salemi,Bastian Schlich, Alexander Schmitz,
Volker Gruhn, Lothar Schöpe, Ursula Wellen
Endbericht der Projektgruppe Com42Bill (PG 411)
März 2003

/136/ Klaus Alfert
Vitruv: Specifying Temporal Aspects of Multimedia Presentations —
A Transformational Approach based on Intervals
April 2003

/137/ Klaus Alfert, Jörg Pleumann, Jens Schröder
A Framework for Lightweight Object-Oriented Design Tools
April 2003

/138/ K. Alfert, A. Fronk , Ch. Veltmann (Hrsg.)
Stefan Borggraefe, Leonore Brinker, Evgenij Golkov, Rafael Hosenberg, Bastian Krol, Daniel Mölle, Markus Niehammer, Ulf Schellbach, Oliver Szymanski, Tobias Wolf, Yue Zhang
Endbericht der Projektgruppe 415: Konzeption und Implementierung eines digitalen und hypermedialen Automobilcockpits (HyCop)
Mai 2003

/139/ Volker Gruhn, Malte Hülder, Sami Beydeda (Hrsg.)
Endbericht der Projektgruppe 409: Entwicklung von ortsbasierten Diensten für UMTS-Mobilfunkgeräte (mCube)
Mai 2003

/140/ Ernst-Erich Doberkat
Congruences for Stochastic Relations
Juli 2003

/141/ Marion Kamphans, Sigrid Metz-Göckel, Anja Tigges
Wie Geschlechteraspekte in die digitalen Medien integriert werden können – das BMBF-Projekt „MuSofT"
September 2003

/142/ Ernst-Erich Doberkat
Semi-Pullbacks for Stochastic Relations over Analytic Spaces
Januar 2004

/143/ Volker Gruhn, Lothar Schöpe (Hrsg.)
1. Workshop des Verbundforschungsprojektes Mobile Spedition im Web – SpiW
Oktober 2002

/144/ Ernst-Erich Doberkat
Stochastic Relations Interpreting Modal Logic
Oktober 2003

/145/ Alexander Fronk, Ernst-Erich Doberkat, Johannes Bergemann, Ulrich-Walter Gans
Ein interdisziplinäres methodisches Vorgehen zur Gestaltung webbasierter Studieneinheiten für die Altertumswissenschaften
November 2003

/146/ Ernst-Erich Doberkat
Factoring Stochastic Relations
Januar 2004

/147/ Ernst-Erich Doberkat
Characterizing the Eilenberg-Moore Algebras for a Monad of Stochastic Relations
March 2004

/148/ Ernst-Erich Doberkat, Gregor Engels, Corina Kopka (Hrsg.) Abschlussbericht des Projektes "MuSofT – Multimedia in der SoftwareTechnik"
April 2004

/149/ Ernst-Erich Doberkat
Derandomizing probabilistic semantics through Eilenberg-Moore algebras for the Giry monad
July 2004

/150/ Alexander Fronk, Jens Schröder (Hrsg.)
Armin Bruckhoff, Stephan Eisermann, Kai Gutberlet, Michél Kersjes, André Kupetz, Christian Mocek, Michael Nöthe, Michael Pflug, René Schönlein, Semih Sevinç, Daniel Unger, Sven Wenzel, Jan Wessling
Abschlussbericht der PG 444: Eclipse Framework for Editing Complex Three-Dimensional Software Visualizations
August 2004